University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models

Nicholas John Dingle

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, October 2004

Abstract

Response time quantiles reflect user-perceived quality of service more accurately than mean or average response time measures. Consequently, on-line transaction processing benchmarks, telecommunications Service Level Agreements and emergency services legislation all feature stringent 90th percentile response time targets.

This thesis presents techniques and tools for extracting response time densities, quantiles and moments from large-scale models of real-life systems. This work expands the applicability, capacity and specification power of prior work, which was hitherto focused on the analysis of Markov models which only support exponential delays.

Response time densities or cumulative distribution functions of interest are computed by calculating and subsequently numerically inverting their Laplace transforms. We develop techniques for the extraction of response time measures from Generalised Stochastic Petri Nets (GSPNs) and Semi-Markov Stochastic Petri Nets (SM-SPNs). The latter is our proposed modelling formalism for the high-level specification of semi-Markov models which support generally-distributed delays.

The techniques presented improve dramatically on the state-space capacity of previous work in two ways. Firstly, we use a space-efficient function representation scheme based on the evaluation demands of a numerical Laplace transform inversion algorithm. Secondly, we exploit the processing power and memory capacity of a network of machines to perform calculations in parallel. Hypergraph partitioning is used to minimise the amount of communication between processors whilst ensuring that the computational load is balanced. An alternative approach, based on exact state-level aggregation, is also described.

Finally, we describe an extended Continuous Stochastic Logic (eCSL) for the formulation of performance queries for high-level models in a concise and rigorous manner.

Response time and scalability results which have been produced on a range of architectures (including workstation clusters and parallel computers) are presented for several case studies. Our implementations exhibit good scalability and demonstrate the ability to analyse models with state spaces of $O\left(10^7\right)$ states and above.

Acknowledgements

I would like to thank the following people:

- My supervisor, Dr. William Knottenbelt, for his help and enthusiasm throughout the course of my research.
- My friends and family for their love and support, especially during the writing of this thesis.
- The members of the Analysis, Engineering, Simulation and Optimisation of Performance (AESOP) research group. In particular: Ashok Argent-Katwala, Susanna Au-Yeung, Jeremy Bradley, Tony Field, Uli Harder, Peter Harrison, David Thornley, Aleks Trifunović and Harf Zatschler.
- Keith Sephton and the Imperial College Parallel Computing Centre for the use of the Fujitsu AP3000 parallel computer.
- The London e-Science Centre for the use of the Viking Beowulf cluster.
- The Engineering and Physical Sciences Research Council (EPSRC) for providing me with the funding to do my PhD.

Time is everything: five minutes makes the difference between victory and defeat.

Vice Admiral Lord Nelson

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Contents

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 Real World Examples . . . 1

1.1.2 Performance Modelling . . . 4

1.2 Aims and Objectives . . . 7

1.3 Contributions . . . 7

1.4 Outline . . . 10

1.5 Statement of Originality and Publications . . . 13

1.6 Notation . . . 15

2 Background 18

2.1 Stochastic Processes . . . 18

2.1.1 Discrete-Time Markov Chains . . . 20

2.1.2 Continuous-Time Markov Chains . . . 22

2.1.3 Semi-Markov Processes . . . 24

2.1.4 Classical Iterative Techniques for Steady-state Analysis . . . 25

2.2 High-level Modelling Formalisms . . . 26

2.2.1 Petri Nets . . . 26

2.2.2 Stochastic Petri Nets . . . 28

2.2.3 Generalised Stochastic Petri Nets 29
2.2.4 Semi-Markov Stochastic Petri Nets 31
2.2.5 Stochastic Process Algebras 36
2.2.6 Queueing Networks 38
2.3 Laplace Transforms 40
2.3.1 Properties 41
2.3.2 Laplace Transform Inversion 43

3 Passage Times in Markov Models 49

3.1 The Laplace Transform Method for CTMCs 49
3.2 Extension to GSPNs 53
3.2.1 Example of GSPN Analysis 54
3.3 Uniformization 57
3.3.1 Uniformization for Transient Analysis of CTMCs 57
3.3.2 Uniformization for Passage Time Analysis of CTMCs 58
3.4 Comparison of Methods 60
3.4.1 Models Studied 63
3.4.2 Discussion 64
3.5 Passage Times in Stochastic Process Algebras 66
3.5.1 Example of SPA Analysis 67
3.6 Estimation of Passage Time Densities and Distributions From Their Moments 68
3.6.1 Moment Calculation for CTMCs 69
3.6.2 Extension to GSPNs 71
3.6.3 Distribution Estimation from Moments 71

4 Passage Times in Semi-Markov Models **80**

4.1 Efficient Representation of General Distributions 81

4.2 The Laplace Transform Method for SMPs 82

4.3 Iterative Passage Time Analysis 83

4.3.1 Technical Overview 84

4.3.2 Example Passage Time Results 85

4.3.3 Practical Convergence of the Iterative Passage Time Algorithm 90

4.4 Iterative Transient Analysis 92

4.4.1 Technical Overview 93

4.4.2 Example Transient Results 98

4.4.3 Practical Convergence of the Iterative Transient Algorithm . . 99

4.5 Estimation of Passage Time Densities and Distributions From Their Moments 101

4.5.1 Moment Calculation 102

4.5.2 Example Results 104

5 Techniques for Analysing Large Models **107**

5.1 Sparse Matrix Partitioning Strategies 110

5.1.1 Graph Partitioning 113

5.1.2 Hypergraph Partitioning 115

5.1.3 Evaluation 117

5.2 State-level Aggregation for Semi-Markov Processes 121

5.2.1 Aggregation Algorithm 123

5.2.2 State-ordering Strategies 126

5.2.3 State-selection Algorithms 128

5.2.4 Comparing Aggregation Strategies . . . 129
5.2.5 Comparing Models of Different Size . . . 133
5.2.6 Parallel Aggregation . . . 134

6 Implementations **140**

6.1 DNAmaca . . . 141
6.1.1 Model Specification Language . . . 143
6.1.2 Model Description . . . 143
6.1.3 Solution Control . . . 146
6.1.4 Performance Measure Specification . . . 146
6.1.5 State-space Generator . . . 147
6.1.6 Functional Analyser . . . 150
6.1.7 Steady-state Solver . . . 151
6.1.8 Sparse Matrix Representation . . . 151
6.1.9 Performance Analyser . . . 154
6.2 HYDRA . . . 154
6.2.1 Input Language Augmentation . . . 156
6.2.2 State Generator and Steady-state Solver . . . 157
6.2.3 Matrix Uniformization and Transposition . . . 158
6.2.4 Hypergraph Partitioner . . . 158
6.2.5 Uniformization-based Passage Time and Transient Analyser . 158
6.3 SMCA . . . 160
6.3.1 Input Language Augmentation . . . 160
6.3.2 State-space Generator . . . 162
6.3.3 Steady-state Solver . . . 162

6.3.4 Example SMP Steady-state Analysis 163

6.4 SMARTA 166

6.4.1 Tool Architecture 166

6.4.2 Implementation of the Parallel Iterative Algorithm 168

7 Extended Continuous Stochastic Logic 172

7.1 CSL 173

7.1.1 Formal CSL semantics 174

7.1.2 Opportunities for Enhancing CSL 174

7.2 eCSL 175

7.2.1 The Syntax of eCSL 176

7.2.2 Examples of eCSL Formulae 177

7.2.3 Formal Stochastic Semantics of eCSL 178

8 Numerical Results for Very Large Markov and Semi-Markov Models 181

8.1 Very Large Markov Models 181

8.1.1 Flexible Manufacturing System 182

8.1.2 Tree-like Queueing Network 183

8.1.3 HYDRA Scalability 186

8.2 Very Large Semi-Markov Models 189

8.2.1 Transient Analysis 189

8.2.2 Passage Time Analysis 190

8.2.3 SMARTA Scalability 193

9 Conclusion 199

9.1 Summary of Achievements 199

9.2 Applications 202

9.3 Future Work 203

A Models **205**

A.1 Courier Communications Protocol 205

A.2 Flexible Manufacturing System 212

A.3 Tree-like Queueing Network 217

A.4 Voting Model 221

A.5 Web Content Authoring System 224

B Semi-Markov Process Aggregation Algorithm **228**

B.1 Aggregation Functions 230

B.2 Utility functions 231

Bibliography **232**

List of Tables

1.1 Response time constraints for transactions in the TPC-C benchmark [120]. . . . 2

1.2 Some common probability density and cumulative distribution functions. 17

3.1 Comparison of run-time in seconds for uniformization, Laplace transform inversion and simulation passage time analysis. . . . 63

3.2 Run-time in seconds for the Laplace transform inversion method on GSPN state-spaces without vanishing state elimination. . . . 63

3.3 Comparison of run-times in seconds for GLD approximation and full Laplace transform-based passage time solution. . . . 78

5.1 Run-times for hypergraph partitioned and row-striped parallel sparse matrix–vector multiplication for the analysis of 165 s-points in the $249\,760$ state Voting model using the iterative algorithm of Chapter 4. 117

5.2 Run-times for hypergraph partitioned and row-striped parallel sparse matrix–vector multiplication for the $1\,639\,440$ state FMS model using uniformization (512 multiplications). . . . 118

5.3 Speedup figures for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of 165 s-points in the $249\,760$ state Voting model. . . . 118

5.4 Speedup figures for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of the $1\,639\,440$ state FMS model using uniformization. . . . 118

5.5 Percentage of state-space which can be aggregated without requiring information stored on remote processors. Shown for differing numbers of partitions in a number of different sized models. 139

6.1 Some common transition delay density functions and their corresponding Laplace transforms. 162

8.1 Communication overhead in the queueing network model with six customers (left) and interprocessor communication matrix (right) for each processor in a 4 processor decomposition. 184

8.2 Per-iteration communication overhead for various partitioning methods for the queueing network model with 27 customers on 16 processors. 184

8.3 Run-time, speedup (S_p), efficiency (E_p) and per-iteration communication overhead for p-processor passage time density calculation in the FMS model with $k = 7$. Results are presented for an AP3000 distributed-memory parallel computer and a PC cluster. 186

8.4 Run-time, speedup and efficiency of performing hypergraph-partitioned sparse matrix–vector multiplication across 1 to 32 processors. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 194

8.5 Run-time, speedup and efficiency using 32 slave processors divided into various different size sub-clusters. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 194

A.1 Number of states generated by the Voting model SM-SPN in terms of the number of voters (CC), polling units (MM) and central voting units (NN). 222

A.2 Number of states generated by the Web-server SM-SPN in terms of the number of clients (RR), authors (WW), parallel web servers (SS) and write-buffers (BB). 225

List of Figures

2.1 An example Place-Transition net (left) and the result of firing transition t_1 (right). 27

2.2 An example Generalised Stochastic Petri Net (GSPN) [48]. 30

2.3 An example Semi-Markov Stochastic Petri Net (SM-SPN) [25]. . . . 35

2.4 An example PEPA model. 38

2.5 An example open queueing network. 38

2.6 Algorithm for automatically determining scaling parameters [70]. . . 47

3.1 The Simple GSPN model. 55

3.2 The reachability graph of the Simple GSPN model. 55

3.3 Numerical and simulated response time densities for the Simple model for time taken from markings where $M(p_1) > 0$ to markings where $M(p_2) > 0$. Here the transition rate parameters are $r = 2$ and $v = 5$. . 56

3.4 Numerical and simulated (with 95% confidence intervals) passage time densities for time taken from the initiation of a transport layer transmission to the arrival of an acknowledgement packet in the Courier model. 60

3.5 Numerical and simulated (with 95% confidence intervals) density for the time taken to produce a finished part of type $P12$ starting from states in which there are $k = 6$ unprocessed parts of types $P1$ and $P2$ in the FMS model. 61

3.6 Numerical and simulated (with 95% confidence intervals) passage time densities for the cycle-time in a tree-like queueing network with 15 customers. 61

3.7 GSPN passage time density calculation pipeline [48]. 62

3.8 The PEPA description for the generalised active badge model with N rooms and M people [23]. 67

3.9 Passage time density of the time taken for the first person to move from room 1 to room 6 in the 3-person Active Badge model. 68

3.10 The branching Erlang model. 75

3.11 The approximate Courier model passage time density function produced by the GLD method compared with the exact result. 75

3.12 The approximate Courier model cumulative distribution function produced by the GLD method compared with the exact result. 76

3.13 The approximate Erlang model passage time density function produced by the GLD method compared with the exact result. 76

3.14 The approximate Erlang model cumulative distribution function produced by the GLD method compared with the exact result. 77

3.15 The branching Erlang model cumulative distribution function produced by the GLD method compared with the bounds produced by the Win-Moments tool. 77

4.1 Numerical and simulated (with 95% confidence intervals) density for the failure mode passage in the Voting model system 1 ($2\,081$ states). . 86

4.2 Cumulative distribution function and quantile for the failure mode passage in the Voting model system 1 ($2\,081$ states). 86

4.3 Numerical and simulated (with 95% confidence intervals) density for the time taken to process 45 reads and 22 writes in the Web-server model system 1 ($107\,289$ states). 87

4.4 Cumulative distribution function and quantile for the time taken to process 45 reads and 22 writes in the Web-server model system 1 (107 289 states). 87

4.5 Numerical and simulated (with 95% confidence intervals) density for the time taken to process 175 voters in the Voting model system 7 (1.1 million states). 88

4.6 Cumulative distribution function and quantile for the time taken to process 175 voters in the Voting model system 7 (1.1 million states). . . . 88

4.7 Average number of iterations to converge per s point for two different values of ε over a range of model sizes for the iterative passage time algorithm. 90

4.8 Average time to convergence per s point for two different values of ε over a range of model sizes for the iterative passage time algorithm. . 91

4.9 Average number of iterations per unit time over a range of model sizes for the iterative passage time algorithm. 91

4.10 A simple two-state semi-Markov process. 95

4.11 Example iterations towards a transient state distribution in a system with successive exponential and deterministic transitions. 95

4.12 Example iterations towards a transient state distribution in a system with successive deterministic and exponential transitions. 96

4.13 Where numerical inversion performs badly: transient state distribution in a system with two deterministic transitions. 96

4.14 The effect of adding randomness: transient state distribution of the two deterministic transitions system with a initial exponential transition added. 97

4.15 Transient and steady-state values in system 1, for the transit of 5 voters from the initial marking to place p_2. 97

4.16 Average number of iterations to converge per s point for two different values of ε over a range of model sizes for the iterative transient algorithm. 100

4.17 Average time to convergence per s point for two different values of ε over a range of model sizes for the iterative transient algorithm. 100

4.18 Average number of iterations per unit time over a range of model sizes for the iterative transient algorithm. 101

4.19 A simple four-state semi-Markov model (SMFour). 105

4.20 The SMFour model passage time density function produced by the GLD method compared with the exact result. 105

4.21 The SMFour model cumulative distribution function produced by the GLD method compared with the exact result. 106

5.1 A 16×16 non-symmetric sparse matrix $\mathbf{A}$ [50]. 110

5.2 The 4-way row-striped partition of the matrix $\mathbf{A}$ in Fig. 5.1 and the corresponding partition of the vector $\mathbf{x}$. 111

5.3 The 4-way 2D checkerboard partition of the matrix $\mathbf{A}$ in Fig. 5.1 (with random asymmetric row and column permutation) and the corresponding partition of the vector $\mathbf{x}$. 111

5.4 The 4-way graph partition of the matrix $\mathbf{A}$ in Fig. 5.1 and the corresponding partition of the vector $\mathbf{x}$ [50]. 114

5.5 A graph (left) and a hypergraph (right) [121]. 115

5.6 The 4-way hypergraph partition of the matrix $\mathbf{A}$ in Fig. 5.1 and the corresponding partition of the vector $\mathbf{x}$ [50]. 116

5.7 Speedup for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of 165 s-points in the $249\,760$ state Voting model. 119

5.8 Speedup for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of the 1 639 440 state FMS model using uniformization. 119

5.9 Reducing a complete 4 state graph to a complete 3 state graph. 122

5.10 Aggregating sequential transitions in an SMP. 123

5.11 Aggregating branching transitions in an SMP. 124

5.12 The three-step removal of a cycle from an SMP. 124

5.13 Complete aggregation of a 2 081 state semi-Markov system to two states.127

5.14 Transition matrix density for the 2 081 state model for four different state-selection algorithms. 130

5.15 Computational cost (in terms of sequential, branching and cycle reduction operations) for the 2 081 state model for four different state-selection algorithms. 130

5.16 Transition matrix density over two different model sizes and two different state-selection algorithms. 131

5.17 Computational complexity over two different model sizes and two different state-selection algorithms. 131

5.18 Computational complexity for systems with up to 541 280 states; fewest-paths-first algorithm only. 132

5.19 Transition matrix density for systems with up to 541 280 states; fewest-paths-first algorithm only. 132

5.20 Transition matrix for the 2 081 state Voting model. 135

5.21 Hypergraph bi-partitioned transition matrix for the 2 081 state Voting model. 136

5.22 Aggregated hypergraph-partitioned transition matrix for the 2 081 state Voting model (contains 374 states). 137

6.1 DNAmaca tool architecture [87]. 142

6.2 Breadth-first search algorithm for state-space exploration [88]. 148

6.3 DNAmaca's sparse matrix representation scheme [87]. 152

6.4 HYDRA tool architecture. 155

6.5 A three-state SM-SPN. 163

6.6 The SMCA input file for the SM-SPN in Fig. 6.5. 164

6.7 The SMCA performance analyser output for the model file in Fig. 6.6. 165

6.8 SMARTA: Semi-Markov Passage Time Analyser. 167

6.9 Parallel iterative passage time calculation algorithm for slave processor i. 169

7.1 An example of a transient constraint $\vec{m} \models \mathcal{T}_{R_p}^{R_t}(\Psi)$ which is satisfied by a transient distribution in the shaded area. 177

8.1 Numerical and simulated (with 95% confidence intervals) passage time densities for the time taken to produce a finished part of type $P12$ starting from states in which there are $k = 7$ unprocessed parts of types $P1$ and $P2$. 182

8.2 Transposed $\mathbf{P}'$ matrix (left) and hypergraph-partitioned matrix (right) for the tree-like queueing network with 6 customers (5 544 states). . . 183

8.3 Numerical and analytical cycle time densities for the tree-like queueing network of Fig. A.3 with 27 customers ($10\,874\,304$ states). 185

8.4 Distributed run-time for the FMS model with $k = 7$ on the AP3000 and a PC cluster. 187

8.5 Speedup for the FMS model with $k = 7$ on the AP3000 and a PC cluster. 187

8.6 Efficiency for the FMS model with $k = 7$ on the AP3000 and a PC cluster. 188

8.7 Transient and steady-state values in Voting system 3, for the transit of 5 voters from the initial marking to place p_2. 190

8.8 Numerical and simulated (with 95% confidence intervals) density for the time taken to process 300 voters in the Voting model system 8 (10.9 million states). 191

8.9 Cumulative distribution function and quantile of the time taken to process 300 voters in the Voting model system 8 (10.9 million states). . . 191

8.10 Numerical and simulated (with 95% confidence intervals) density for the time taken to process 100 reads and 50 page updates in the Web-server model system 6 (15.4 million states). 192

8.11 Cumulative distribution function and quantile of the time taken to process 100 reads and 50 page updates in the Web-server model system 6 (15.4 million states). 192

8.12 Run-time of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 195

8.13 Speedup of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 195

8.14 Efficiency of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 196

8.15 Run-time of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 196

8.16 Speedup of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the $249\,760$ state Voting model for 165 s-points. 197

8.17 Efficiency of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster. 197

A.1 The Courier communications protocol GSPN model [125]. 206

A.2 The GSPN model of a Flexible Manufacturing System [41]. 212

A.3 The tree-like queueing network [68, 70]. 216

A.4 The Voting Model SM-SPN [24]. 221

A.5 The Web-server Model SM-SPN [28, 29]. 225

B.1 The $aggregate_smp$ function 228

Chapter 1

Introduction

1.1 Motivation

A fast response time is an important performance criterion for almost all computer-communication and transaction processing systems. Examples of systems with stringent response time requirements include stock market trading systems, mobile communication systems, web servers, database servers, manufacturing systems, communication protocols and communications networks. Typically, response time targets are specified in terms of quantiles (percentiles). For example, in a mobile messaging system it might be required that "there should be a 95% probability that a text message will be delivered within 3 seconds". To further illustrate the importance of response time quantiles in the real world we will consider three application areas in detail, viz. benchmark suites, Service Level Agreements and emergency services legislation.

1.1.1 Real World Examples

Benchmarks

The Transaction Processing Performance Council (TPC) benchmarks [120] were conceived to compare different implementations of large-scale on-line transaction processing (OLTP) systems in a consistent way. A range of benchmarks are available,

Transaction Type	Minimum Percentage of Mix	Minimum Keying Time (sec)	**90th Percentile Response Time Constraint (sec)**	Minimum Mean of Think Time Distribution (sec)
New-Order	n/a	18.0	**5.0**	12.0
Payment	43.0	3.0	**5.0**	12.0
Order-Status	4.0	2.0	**5.0**	10.0
Delivery	4.0	2.0	**5.0**	5.0
Stock-Level	4.0	2.0	**20.0**	5.0

Table 1.1. Response time constraints for transactions in the TPC-C benchmark [120].

each of which is suitable for different applications including transaction processing, decision support, business reporting and e-Commerce [120]. For example:

> TPC Benchmark™ C (TPC-C) is an OLTP workload. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments … The performance metric reported by TPC-C is a “business throughput” measuring the number of orders processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint.

The TPC-C workload consists of five different types of transaction, three of which have strict response time requirements (New-Order, Payment and Order-Status transactions) and two where these requirements are more relaxed (Delivery and Stock-Level transactions). The response time constraints on each type of transaction are summarised in Table 1.1.

In order for a set of results to be considered compliant with the TPC-C benchmark, a full disclosure report must be supplied. Amongst other things, this must include [120]:

> The numerical quantities listed below must be summarised near the beginning of the Full Disclosure report [including] … ninetieth percentile, average and maximum response times for the New-Order, Payment, Order-

> Status, Stock-Level, Delivery (deferred and interactive) and Menu transactions ... Response Time frequency distribution curves ... must be reported for each transaction type.

Service Level Agreements

Service Level Agreements (SLAs) exist as contracts between service providers and their customers [44, 52]. For example, an e-commerce site may have an SLA with the company which hosts its website, or two Internet Service Providers (ISPs) may have mutual SLAs to regulate the carrying of each other's traffic. A typical SLA specifies the level of service to be provided (according to metrics such as availability, response time, latency, packet loss and so forth) and how much this will cost, as well describing what financial penalties will be incurred if this level is not met. It should also describe what level of technical support will be given to the customer in the event of problems.

Usually, the main metric of interest to customers is the availability of the provider's service (e.g. network or servers). However, customers (particularly those involved in web-commerce) often require response time guarantees as well [52]:

> Availability is one metric used to guarantee network and application uptime, but according to a study conducted by Cahners most business executives rank response time as the second most important service factor after availability. Sluggish response time is a major problem facing e-commerce. Potential customers who cannot get through to a Web site quickly get disgusted and bored, and may even give up on using the Web for commerce.

More informally, the Engineering and Physical Sciences Research Council (EPSRC) uses response time quantiles to offer quality of service guarantees to academics applying for research funding. In acknowledgement letters sent out on receipt of research grant proposals, it is stated that:

> The EPSRC aims to notify 90% of all proposers of the outcome of their proposals within 26 weeks of receipt.

Emergency Services

Response time percentiles are also used by governmental organisations when measuring the effectiveness of emergency services. Indeed, in Ontario, Canada, it is a legal requirement to report 90th percentile response times for ambulance services [43, 104, 119]:

> The 90th Percentile Response Time is a legislated requirement that was established by the Ministry of Health to be used as a benchmark to measure the efficiency and effectiveness of a land ambulance service, based on the 90th Percentile Response Time from 1996 [119].

By way of example, in [119] it is reported that:

> In 1996, the 90th Percentile Response Time for the services within Leeds & Grenville was 17:45 (mm:ss). In the subsequent years, from 1997 to 2000, the ambulance services in Leeds & Grenville were unable to meet the time established in 1996. However, in 2001, Leeds & Grenville EMS was able to meet and exceed the 1996 90th Percentile Response Time with a time of 17:26 (mm:ss).

Similar reporting takes place in Australia [9] and San Francisco [113]. In the UK, the London Ambulance Service aims to have an ambulance at the scene of 75% percent of life-threatening incidents within 8 minutes [97] while the National Health Service aims to see 90% of accident and emergency patients within 4 hours [42].

1.1.2 Performance Modelling

As can be seen from the above examples, it is important to ensure that systems will meet quality of service targets expressed in terms of response time quantiles. Ideally, it should be possible to determine whether or not this will be the case at design time. This can be achieved through the modelling and analysis of the system in question. Such analysis is usually conducted by capturing the behaviour of the system with a

formal model; that is, identifying the possible states the system may be in and the way in which it can move between these states. The concept of time can be introduced by associating delays with the state transitions. The result is that a certain amount of time will be spent in a state before moving to another, and we term this the *state sojourn time*. When the choice of the next state depends only on the current state and state sojourn times are random numbers sampled from the negative exponential distribution, we call such a model a *continuous-time Markov chain*.

As specifying every state and transition in the state space of a complex model of a real-life system is infeasible, high-level formalisms such as stochastic Petri nets [15], stochastic process algebras [75] and queueing networks [78] can be employed. These permit a succinct description of the model from which a Markov chain can automatically be extracted and then solved for performance measures of interest. From the equilibrium (steady-state) probability distribution of the model's underlying Markov chain, standard resource-based performance measures, such as mean buffer occupancy, system availability and throughput, and *expected* values of various sojourn times can be obtained. There is a large body of previous work on the efficient calculation of steady-state probabilities in large Markov chains, including parallel [16, 32, 88] and disk-based [46, 89, 93] implementations, as well as those which employ implicit state space representation techniques [38, 47, 74, 94]. Steady-state measures allow the answering of questions such as: "What is the probability that the system will be in a failure state in the long run?" and "What is the average utilisation of this resource?".

The focus of this thesis, however, is on the harder problem of calculating full response time densities in very large Markov models and semi-Markov models (a generalisation of Markov models in which state sojourn times can have an arbitrary distribution). As we have seen, the answers to response time questions provide greater insight into whether or not a system meets its user requirements than steady-state probabilities. In the context of high-level models, response times can be specified as *passage* times in the model's underlying Markov or semi-Markov chain – that is, the time taken to enter any one of a set of target states having started from a specified set of source states.

In the past, numerical computation of analytical passage time densities has proved

prohibitively expensive except in some Markovian systems with restricted structure such as overtake-free tree-like queueing networks [68]. However, with the advent of high-performance parallel computing and the widespread availability of PC clusters, direct numerical analysis of Markov chains has now become a practical proposition. There are two main analytical methods for computing first passage time (and hence response time) densities in Markov chains: those based on Laplace transforms and their inversion [70] and those based on uniformization [99, 102, 105]. The former has wider application to semi-Markov processes (with generally-distributed state holding-times) but is less efficient than uniformization when restricted to Markov chains.

In general, the probability density function of the time taken to move from a set of source states to a set of target states is calculated by convolving the state-holding time functions along all possible paths between the two sets of states. To convolve two functions together directly requires the evaluation of an integral, and the convolution across a path n states long requires the evaluation of an $(n-1)$ dimensional integral. To perform such a calculation for large values of n (perhaps in the millions) would therefore be impractical. Instead, we make use of Laplace transforms, which uniquely map a real-valued function (e.g. a probability density function) to a function of a complex variable. We do this as we wish to exploit the convolution property of Laplace transforms, which states that the Laplace transform of the convolution of two functions is the product of the functions' individual Laplace transforms. Once the Laplace transform of the passage time measure has been calculated it is possible to retrieve the corresponding density function using a process known as *Laplace transform inversion*. A number of numerical techniques are available to accomplish this.

Although all state holding-times in Markov models are exponentially distributed, this does not make the direct calculation of their convolutions significantly easier as the rate parameters of the state holding-time distributions will usually be different for different states. An alternative technique known as uniformization can, however, be employed. This transforms the model's underlying continuous-time Markov chain with different rates out of the states into an equivalent one where all delay rate parameters are identical. The passage time density across any number of these states can therefore

be calculated easily because the convolution of exponential delays with the same rate parameter is simply an Erlang distribution.

As semi-Markov processes do not have identically distributed state holding-time functions, uniformization cannot be applied to calculate passage time measures in such processes. Very little work has been done on the problem of calculating passage time densities and distributions in semi-Markov models, and what has been done is limited to applying analytical techniques to models with small state spaces (of the order of 10^3 to 10^4 states) [64, 98].

1.2 Aims and Objectives

The aims and objectives of this thesis are:

- To develop algorithms for the calculation of passage time densities and quantiles and transient state distributions in semi-Markov models.
- To investigate techniques for the passage time analysis of Markov and semi-Markov models with very large state-spaces. We will focus on two methods. The first is partitioning the state-spaces across a number of processors to conduct analysis in parallel. The second is an exact aggregation strategy for reducing the state-spaces of very large models prior to performing analysis.
- To develop a formal language for specifying performance questions on high-level models.
- To develop parallel tools for the analysis of very large Markov and semi-Markov which implement our passage time and transient algorithms.

1.3 Contributions

This thesis presents techniques and tools for the extraction of passage time densities, quantiles and moments from very large Markov and semi-Markov models. The work

presented expands the applicability, capacity and specification power of prior work, which was hitherto focused mainly on the analysis of Markov models.

We extend existing work on the extraction of passage time densities and quantiles from Markov models to the analysis of Generalised Stochastic Petri Nets (GSPNs) and Semi-Markov Stochastic Petri Nets (SM-SPNs), the latter being our proposed modelling formalism for the high-level specification of semi-Markov models. The general approach we take is to compute the Laplace transform of the passage time density by convolving the Laplace transforms of the state holding-time functions together. This is then numerically inverted to yield the required passage time measure.

Previous attempts to perform passage time analysis of semi-Markov processes (SMPs) have foundered on the complexity of maintaining a symbolic representation of the state holding-time functions under composition which limits the size of models that can be analysed. We have overcome this by developing a representation scheme based on the evaluation demands of the Laplace transform inversion algorithms which requires constant space even when these functions are convolved or added together.

We have devised an iterative algorithm to calculate the Laplace transform of the passage time quantity of interest, the kernel of which is repeated sparse matrix–vector multiplication. We have also extended this approach to the efficient calculation of transient state distributions in SMPs. For very large models, however, it is not possible to maintain the transition matrix in the memory of a single machine. Consequently, we take advantage of the combined memory capacity and computational power of a group of machines to divide the matrix across multiple processors and perform the calculations in parallel. This requires updated vector elements to be exchanged after each iteration and, in order for the parallelisation to be efficient, it is vital that the amount of communication be minimised whilst ensuring that the computational load is balanced. To achieve this we use hypergraph partitioning (traditionally used in the VLSI domain). Experimentation shows that this offers significant run-time benefits over naïve row-striped (linear) partitioning, particularly on machines or clusters with relatively slow interconnection networks.

An alternative to parallel computation for the analysis of very large models is to at-

tempt to reduce the size of the model's state-space by aggregating states together. An exact state-level aggregation algorithm for semi-Markov processes (originally presented in [21]) is described and its behaviour when applied to large models is investigated. As this algorithm operates on a single state at a time it lends itself to a parallel implementation, and the issues which arise when implementing this are discussed.

As well as specifying high-level models in a formal manner, it is also beneficial to ask questions about them in a formal way. There is a large body of prior work on the use of Continuous Stochastic Logic (CSL) to check CTMCs and SMPs for steady-state and passage time properties [10, 11, 13, 14, 85, 98]. CSL operates at the state-transition level of the model's underlying Markov or semi-Markov chain, which requires the reasoning about paths through the state-space. A more natural mode of expression for the performance modeller is to pose questions at the level of the high-level model – in the case of Petri nets, performance queries can be posed more easily in terms of the number of tokens on places. We therefore propose an extended Continuous Stochastic Logic (eCSL) which permits the formulation of steady-state, transient or passage time queries directly at the SM-SPN model level. These queries are then answered either by traditional steady-state analysis or by application of the iterative passage time or transient algorithm.

We have implemented three tools based on the algorithms and techniques described in this thesis. These build on the input language specification, probabilistic state-space generation and steady-state solution techniques of DNAmaca [87], a steady-state analyser for Markov chains. The first tool we present, HYDRA (HYpergraph-based Distributed Response Time Analyser), calculates passage time densities and transient distributions in large Markov models through the use of uniformization. It employs hypergraph partitioning to permit the efficient parallel analysis of very large state spaces. Next, we describe SMCA (Semi-Markov Chain Analyser), a semi-Markov extension of DNAmaca for the steady-state analysis of large semi-Markov chains. Finally, we present SMARTA (Semi-MArkov Response Time Analyser), which implements our iterative passage time algorithm for the analysis of very large semi-Markov models. SMARTA uses numerical Laplace transform inversion to calculate the required pas-

sage time measure. It has a distributed architecture with a master process which hands out work to groups of slaves. Each group of slaves employs the hypergraph partition of the state graph to carry out its work efficiently in parallel.

In order to demonstrate the capacity and scalability of our implementations, we conduct analysis on a number of large Markov and semi-Markov models of real-life systems. The three Markov models are of a communication protocol, a Flexible Manufacturing System and a tree-like queueing network. The first semi-Markov model is of a distributed electronic voting system with voting booths and central servers, both of which are subject to random failures. The second is a model of a web-content authoring system, where authors publish material on unreliable web-servers for a pool of readers to read. Both of the semi-Markov models are specified using the SM-SPN formalism. We generate a range of state space sizes by altering the initial number of tokens in the high-level models (corresponding to the number of voters or authors, for example).

In order to analyse very large models with 10 million states and above in an acceptable time, the Imperial College Parallel Computing Centre's Fujitsu AP3000 (60 Ultra-SPARC 300MHz processors with 256MB RAM interconnected by a 2D wraparound mesh network with wormhole routing and a peak throughput of 520Mbps) and the London eScience Centre's Beowulf cluster (comprised of 64 dual-processor nodes with Intel 2.0Ghz Xeon processors and 2GB of RAM connected by Myrinet with a peak throughput of 2Gbps) were used. Thanks to the use of hypergraph partitioning, the two tools (HYDRA and SMARTA) exhibit good scalability, even on clusters of commodity workstations.

1.4 Outline

The remainder of this thesis is organised as follows:

Chapter 2 describes the background theory to the work presented in this thesis. The general topic of stochastic processes is introduced and then three specific ex-

amples (namely discrete-time and continuous-time Markov chains and semi-Markov chains) are described in detail. As it is impractical to describe complex systems with many thousands of states directly with these low-level processes, a number of high-level modelling formalisms are described from which such state-transition systems can be generated. The three examples considered are Petri nets, process algebras and queueing networks. Finally, Laplace transforms and methods for their numerical inversion are described.

Chapter 3 begins by presenting prior work on the extraction of passage time densities and quantiles from Markov models. Two methods are presented: one based on the use of Laplace transforms and the other making use of uniformization as described above. The two techniques are compared with each other and also with simulation in terms of their run-time performance. We have extended this work to facilitate the extraction of passage time densities from Generalised Stochastic Petri Nets (GSPNs), and our modifications to the previously described Laplace transform method are presented here. A method for the approximation of passage time densities from their moments is also presented and the extraction of passage time measures from stochastic process algebra models is demonstrated.

Chapter 4 presents an iterative algorithm for the calculation of passage time densities and quantiles in semi-Markov models. This is achieved by calculating and numerically inverting the Laplace transform of the passage time quantity of interest. Using the inversion algorithms described in Chapter 2, it is possible to determine in advance at which values of the complex parameter s the Laplace transform of the passage time density or distribution function must be computed in order to perform the inversion. This makes it possible to adopt an efficient storage scheme for the generally distributed state holding-time functions which occur in semi-Markov models. These are convolved together, across all paths leading from source to target states, to calculate the passage time density. By storing their values, and these values of the convolutions, only at the previously-identified s-points, enough information to perform the Laplace transform is available without the complexity of attempting to maintain a full sym-

bolic representation. We present the theoretical background to the iterative algorithm and then describe its implementation. Empirical complexity results are also presented. We also describe an extension of the iterative algorithm to the calculation of transient state distributions in semi-Markov models which improves on the computational effort of existing methods. Finally, we describe the calculation of the moments of passage time quantities in semi-Markov processes.

Chapter 5 describes a number of techniques for analysing very large models. A major difficulty experienced in such analysis is the amount of memory required to store the model's transition matrix. Two ways of surmounting this problem are presented. The first is to partition the matrix across a number of processors and conduct the calculations in parallel; a number of schemes for performing this partitioning are described. The most effective is hypergraph partitioning, which symmetrically permutes the rows and columns of the matrix in such a way that the amount of communication required between the processors involved in minimised whilst computational load is balanced. Experimental results are provided which demonstrate its advantages over direct row-striped partitioning. The second method is to reduce the state-space of the model (and hence the dimension of the transition matrix) by aggregating states. An exact aggregation algorithm for semi-Markov processes is described and a number of different state-selection criteria are evaluated. The issues involved in implementing this algorithm in parallel are also considered.

Chapter 6 describes the implementation of three tools for the steady-state, passage time and transient analysis of very large Markov and semi-Markov models. The Markovian DNAmaca steady-state analyser has been extended with HYDRA (HYpergraph-based Distributed Response-time Analyser), which permits the calculation of passage time densities and transient distributions in very large Markov models through the use of uniformization and hypergraph partitioning. We then describe SMCA (Semi-Markov Chain Analyser), a semi-Markov steady-state analyser also based on DNAmaca. This required alterations to the

input language, state generator and steady-state solver in order to deal with generally-distributed transition firing delays. Finally, SMARTA (Semi-MArkov Response Time Analyser), a scalable parallel pipeline which implements the iterative algorithms of Chapter 4 for passage time and transient analysis, is presented. This builds upon the semi-Markov extension to DNAmaca and the hypergraph-partitioned sparse matrix–vector computations performed by HYDRA to analyse very large semi-Markov models.

Chapter 7 presents a formal language for performance queries using an extended Continuous Stochastic Logic (eCSL). We begin by describing existing Continuous Stochastic Logic and then highlight the areas in which it can be enhanced. We add the ability to reason about multiple start states and transient distributions to eCSL, whilst simplifying the expression of passage time measures. We illustrate our enhancements with a number of example formulae.

Chapter 8 presents numerical results produced using the tools described in Chapter 6. We calculate passage time results for Markov chains with up to 10.8 million states using HYDRA. Transient and passage time density and quantile results produced using SMARTA are displayed for two very large semi-Markov models (with up to 10.9 million and 15.4 million states respectively).

Chapter 9 concludes the thesis by summarising and evaluating the achievements presented and highlighting opportunities for future work.

Appendix A describes the five models used as examples throughout the body of the thesis.

Appendix B describes the aggregation algorithm presented in Chapter 5 more fully.

1.5 Statement of Originality and Publications

I declare that this thesis was composed by myself, and that the work that it presents is my own except where otherwise stated.

The following publications arose from work conducted during the course of this PhD. The chapters in this thesis where material from them appears are indicated below.

- **Journal of Parallel and Distributed Computing** (JPDC) [50] and **International Conference on Parallel and Distributed Processing Techniques and Applications** (PDPTA 2003) [49] consider the uniformization method for passage time density calculation in Markov models and describes the use of hypergraphs as a means to implement the calculations efficiently in parallel. Material from these papers appears in Chapters 3, 5 and 8.

- **Workshop On Software and Performance 2002** (WOSP 2002) [48] presents an extension of earlier work on the use of Laplace transforms to calculate passage time densities in Markov models [70] to cover Generalised Stochastic Petri Nets. The section in Chapter 3 on passage times in GSPNs is taken from this paper.

- **Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems** (PMEO/PDS 2003) [24] presents an iterative algorithm for the calculation of passage time densities and quantiles in semi-Markov processes (SMPs). This algorithm makes use of Laplace transforms and their inversion. Chapter 4 of this thesis presents material from this paper. An extended version of this paper presenting the efficient transient state distribution calculation algorithm of Section 4.4 has been submitted to **Future Generation Computer Systems** [26]. The paper presented at **International Conference on Numerical Solution of Markov Chains** (NSMC 2003) [28] extends the PMEO/PDS work with an efficient parallel implementation using hypergraph partitioning. It also considers the convergence behaviour of the iterative algorithm. The latter aspect was joint work with Jeremy Bradley and Helen Wilson. Material from these papers appears in Chapters 4 and 6, and some of the results in Chapter 8 are also presented in this paper. This paper was selected to appear in a special issue of the **Journal of Linear Algebra and Applications** [29].

- **International Workshop on Petri Nets and Performance Models** (PNPM

2003) [25] presents Semi-Markov Stochastic Petri Nets (SM-SPNs), a high-level formalism from which SMPs can be derived, and describes an extended Continuous Stochastic Logic (eCSL) in which performance questions can asked. Material from this paper appears here in Chapters 2 and 7. This was joint work with Jeremy Bradley.

- **Symposium on Performance Evaluation of Computer and Telecommunication Systems** (SPECTS 2003) [27] considers how the aggregation algorithm for SMPs presented in [21] scales to large systems and investigates the effect different state selection strategies have on its computational complexity. This was joint work with Jeremy Bradley. Work from this paper is presented in Chapter 5.

- **International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems** (MASCOTS 2003) [22], **Workshop on Software and Performance 2004** (WOSP 2004) [6] and **UK Performance Engineering Workshop 2003** (UKPEW 2003) [23] consider the extraction of passage time measures from stochastic process algebra models. Material from these papers appears in Chapter 3.

- **UK Performance Engineering Workshop 2002** (UKPEW 2002) [51] discusses the use of graph-partitioning in an asynchronous parallel steady-state solver for Markov chains. Material from this paper appears in Chapter 6.

- **Workshop on Software and Performance 2004** (WOSP 2004) [8] considers ways in which passage time densities can be approximated from their moments. This was joint work with M.Sc. student Susanna Au-Yeung. Material from this paper appears in Chapter 3.

1.6 Notation

A function $f(x)$ has Laplace transform denoted $f^*(s)$, where s is a complex variable. The Laplace transform of a function $f(x)$ may also be denoted $L\{f(x)\}$, while the

corresponding inverse Laplace transform of $f^*(s)$ is denoted $L^{-1}\{f^*(s)\}$. The real part of s is denoted $\mathrm{Re}(s)$ while the imaginary part is referred to as $\mathrm{Im}(s)$.

If χ is a random variable then its average (expected) value is denoted $E[\chi]$.

Matrices are denoted by bold-face capital letters, while the elements of a matrix are addressed as the corresponding lower-case letter with a subscript denoting the (x, y) coordinate of the desired element; a_{ij} is therefore the (i, j)th element of matrix $\mathbf{A}$. 2D blocks of the matrix are denoted $\mathbf{A}_{ij}$. Vectors are denoted by lower-case bold letters (either Roman or Greek), while individual vector elements are not written in bold type and are addressed with a subscript. Thus, π_n is the nth element of the vector $\boldsymbol{\pi}$.

$f'(x)$ is defined as the first derivative of a function $f(x)$, and similarly $f''(x)$ is that function's second derivative. The nth derivative of $f(x)$ is denoted by $f^{(n)'}(x)$. We denote the Laplace transform of a passage time density from state i into a set of target states $\vec{j}$ as $L_{i\vec{j}}(s)$, and the nth moment of this as $M_{i\vec{j}}(n)$. Similarly, the nth moment of the density of the passage time for a single transition from state i to state k is denoted $m_{ik}(n)$.

We also use shorthand notation to refer to a number of probability distributions throughout this thesis. This is presented in Table 1.2.

Distribution	Description	PDF	CDF
$exp(\lambda)$	Exponential with parameter λ	$\lambda e^{-\lambda t}$	$1 - e^{-\lambda t}$
$uni(a, b)$	Uniform with parameters a and b	$\frac{1}{b-a}$ if $a \leq t \leq b$ 0 elsewhere	0 if $t < a$ $\frac{t-a}{b-a}$ if $a \leq t \leq b$ 1 if $t > b$
$erlang(\lambda, n)$	n-stage Erlang with parameter λ	$\frac{\lambda^n t^{(n-1)} e^{-\lambda t}}{(n-1)!}$	$1 - e^{-\lambda t} \sum_{k=0}^{n-1} \frac{(\lambda t)^k}{k!}$
$gamma(\lambda, n)$	Gamma with parameters λ and n	$\frac{\lambda^n}{\Gamma(n)} t^{(n-1)} e^{-\lambda t}$ where $\Gamma(n) = \int_0^\infty t^{(n-1)} e^{-t}\, dt$	No closed form unless n is integer in which case as $erlang(\lambda, n)$
$det(d)$	Deterministic with delay d	∞ if $t = d$ 0 elsewhere	1 for $t \geq d$ 0 elsewhere
$det(0)$	Immediate	∞ if $t = 0$ 0 elsewhere	1 for $t \geq 0$ 0 elsewhere

Table 1.2. Some common probability density and cumulative distribution functions.

Chapter 2

Background

This chapter presents the background theory underlying the work described in this thesis. A general overview of stochastic processes is provided, before considering Markov and semi-Markov chains in more detail. This is followed by a discussion of high-level modelling formalisms from which Markov or semi-Markov chains can be automatically generated. These are grouped under three headings: stochastic Petri Nets (SPNs), stochastic process algebras (SPAs) and queueing networks. We consider both Markovian [15] and semi-Markovian Petri nets [25, 55, 60, 63, 64, 96, 109], while our discussion of SPAs and queueing networks is confined entirely to Markovian examples. We describe one particular SPA in detail, namely Performance Enhanced Process Algebra (PEPA) [75]. This chapter concludes by describing the theory of Laplace transforms as it relates to the calculation of passage time densities, and also details a number of numerical methods by which they can be inverted.

2.1 Stochastic Processes

At the lowest level, the performance modelling of a system can be accomplished by identifying all possible configurations (or *states*) that the system can enter and describing the ways in which the system can move between those states. This is termed the *state-transition* level behaviour of the model, and the changes in state as time progresses describe a *stochastic process*. In this chapter, we focus on those stochastic pro-

cesses which belong to the class known as *Markov processes*, specifically discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and the more general semi-Markov processes (SMPs).

Consider a random variable χ which takes on different values at different times t. The sequence of random variables $\chi(t)$ is said to be a stochastic process. The different values which members of the sequence $\chi(t)$ can take (also referred to as *states*) all belong to the same set known as the *state-space* of $\chi(t)$.

A stochastic process can therefore be classified by the nature of its state-space and of its time parameter. If the values in the state-space of $\chi(t)$ are finite or countably infinite, then the stochastic process is said to have a *discrete state-space* (and may also be referred to as a *chain*). Otherwise, the state-space is said to be *continuous*. Similarly, if the times at which $\chi(t)$ is observed are also countable, the process is said to be a *discrete time* process. Otherwise, the process is said to be a *continuous time* process. In this thesis, all stochastic processes considered have discrete and finite state-spaces, and we focus mainly on those which evolve in continuous time (although some consideration is also given to the solution of discrete time chains).

A *Markov* process is a stochastic process in which the *Markov property* holds. Given that $\chi(t) = x_t$ indicates that the state of the process $\chi(t)$ at time t is x_t, this property stipulates that:

$$\mathbb{P}\big(\chi(t) = x \mid \chi(t_n) = x_n, \chi(t_{n-1}) = x_{n-1}, \ldots, \chi(t_0) = x_0\big) = \mathbb{P}\big(\chi(t) = x \mid \chi(t_n) = x_n\big)$$
$$t > t_n > t_{n-1} > \ldots > t_0$$

That is, the future evolution of the system depends only on the current state and not on any prior states.

Definition 2.1 *A Markov process is said to be* homogenous *if it is invariant to shifts in time, i.e. [15]:*

$$\mathbb{P}\big(\chi(t+s) = x \mid \chi(t_n+s) = x_n\big) = \mathbb{P}\big(\chi(t) = x \mid \chi(t_n) = x_n\big)$$

2.1.1 Discrete-Time Markov Chains

This section considers Markov processes where state changes are observed at discrete time intervals and with discrete state-spaces; we call these discrete-time Markov chains (DTMCs).

Definition 2.2 *The stochastic process $\chi_n, n = 0, 1, 2, \ldots$ is a* DTMC *if, for $n \in \mathbb{N}_0$ [15]:*

$$\mathbb{P}(\chi_{n+1} = x_{n+1} \mid \chi_n = x_n, \chi_{n-1} = x_{n-1}, \ldots, \chi_0 = x_0) = \mathbb{P}(\chi_{n+1} = x_{n+1} \mid \chi_n = x_n)$$

This describes the one-step transition probabilities of a DTMC – that is, the probability that the DTMC moves from state x_n to state x_{n+1} in a single transition (or *step*). These transition probabilities can be expressed as a matrix $\mathbf{P}$, the elements p_{ij} of which are defined as:

$$p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i) \tag{2.1}$$

given the restriction that $0 \leq p_{ij} \leq 1 \ \forall i, j$ and $\sum_j p_{ij} = 1 \ \forall i$.

Definition 2.3 *A Markov chain is* irreducible *if every state is reachable from every other state in one or more transitions. If this is not the case, the chain is said to be* reducible *[15].*

The states in a Markov chain can be distinguished as being either *recurrent* or *transient*. If $f_j^{(m)}$ is the probability of leaving state j and then first returning to it in m transitions, it follows that the probability of ever returning to state j is:

$$f_j = \sum_{m=1}^{\infty} f_j^{(m)}$$

If $f_j = 1$ then it is certain that we will return to state j at some point in the future and so j is said to be *recurrent*. Otherwise, state j is *transient*.

We can reason about the periodicity of a DTMC as follows:

Definition 2.4 *If a Markov chain can return to state j only at steps $\eta, 2\eta, 3\eta, \ldots$ for $\eta \geq 2$, then state j is* periodic with period η. *Otherwise, state j is* aperiodic *[15].*

From the probability of returning to a state j in m steps, $f_j^{(m)}$, the mean recurrence time M_j of state j (the average number of steps needed to return to j for the first time after leaving it) is defined as:

$$M_j = \sum_{m=1}^{\infty} m f_j^{(m)}$$

A state j is classified as *recurrent null* if $M_j = \infty$, or as *recurrent nonnull* if not.

For DTMCs, we define the probability of being in state j at time m having started in state i at time 0, denoted $\pi_{ij}^{(m)}$, as:

$$\pi_{ij}^{(m)} = \mathbb{P}(\chi_m = j \mid \chi_0 = i)$$

A typical performance measure of interest is the probability that a DTMC is in some state (or set of states) at an arbitrary point in the future, irrespective of the state in which it was initially. This is known as a *steady-state* probability, and the set of the steady-state probabilities of all states in a DTMC is referred to as the *limiting* or *steady-state probability distribution*.

Definition 2.5 *The* limiting *or* steady-state probability distribution$\{\pi_j\}$ *of a DTMC is defined as [15]:*

$$\pi_j = \lim_{m\to\infty} \pi_{ij}^{(m)}$$

Definition 2.6 *The* stationary probability distribution *[117] is defined in terms of* $\mathbf{P}$, *the one-step transition probability matrix of a DTMC, and the vector* $\mathbf{z}$ *whose elements* z_i *denote the probability of being in state* i*. The vector* $\mathbf{z}$ *is a probability distribution; i.e.:*

$$z_i \in \mathbb{R}, \qquad 0 \leq z_i \leq 1 \quad \textit{and} \quad \sum_i z_i = 1$$

$\mathbf{z}$ *is said to be a stationary distribution if and only if* $\mathbf{zP} = \mathbf{z}$.

Theorem 2.1 *In an irreducible and aperiodic DTMC, the steady-state probabilities* $\{\pi_j\}$ *always exist and are independent of the initial state probability distribution. Furthermore, one of the following two situations exists [15]:*

- *all states are transient or all states are recurrent null. In this case,* $\pi_j = 0 \;\forall j$ *and no stationary distribution exists. For this condition to occur the state-space must be infinite.*

- *all states are recurrent nonnull. In this case,* $\pi_j > 0 \;\forall j$ *and* $\{\pi_j\}$ *is given by the stationary probability distribution where:*

$$\pi_j = \frac{1}{M_j}$$

For a finite DTMC with N *states, the values of* π_j *are uniquely determined by the equations:*

$$\sum_i \pi_i p_{ij} = \pi_j \;\; \textit{subject to} \;\; \sum_i \pi_i = 1$$

Or in matrix–vector notation, where $\boldsymbol{\pi}$ *is a vector of probabilities* $\{\pi_1, \pi_2, \ldots \pi_N\}$ *and the matrix* $\mathbf{P}$ *has elements* p_{ij} *as defined in Eq. 2.1:*

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$$

The fact that a DTMC has a stationary probability distribution does not imply that it has a steady-state probability distribution. For example, the irreducible periodic DTMC with one-step transition matrix:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.2}$$

has a stationary distribution $(0.5, 0.5)$ but no steady-state probability distribution.

A finite, irreducible and recurrent nonnull DTMC is termed an *ergodic* chain.

2.1.2 Continuous-Time Markov Chains

There also exists a family of Markov processes with discrete state spaces but whose transitions can occur at arbitrary points in time; we call these continuous-time Markov chains (CTMCs). The definitions above for homogeneity, aperiodicity and irreducibility in DTMCs also hold for CTMCs. An homogenous N-state $\{1, 2, \ldots, N\}$ CTMC has state at time t denoted $\chi(t)$. Its evolution is described by an $N \times N$ generator

matrix $\mathbf{Q}$, where q_{ij} is the infinitesimal rate of moving from state i to state j ($i \neq j$), and $q_{ii} = -\sum_{i \neq j} q_{ij}$.

The Markov property imposes the restriction on the distribution of the sojourn times in states in a CTMC that they must be *memoryless* – the future evolution of the system therefore does not depend on the evolution of the system up until the current state, nor does it depend on how long has already been spent in the current state. This means that the sojourn time ν in any state must satisfy:

$$\mathbb{P}(\nu \geq s + t \mid \nu \geq t) = \mathbb{P}(\nu \geq s) \tag{2.3}$$

A consequence of Eq. 2.3 is that all sojourn times in a CTMC must be exponentially distributed (see [15] for a proof that this is the only continuous distribution function which satisfies this condition). The rate out of state i, and therefore the parameter of the sojourn time distribution, is μ_i and is equal to the sum of all rates out of state i, that is $\mu_i = -q_{ii}$. This means that the density function of the sojourn time in state i is $f_i(t) = \mu_i \, e^{-\mu_i t}$ and the average sojourn time in state i is μ_i^{-1}.

We define the steady-state distribution for a CTMC in a similar manner as for a DTMC. Once again, we denote the set of steady-state probabilities as $\{\pi_j\}$.

Definition 2.7 *In a CTMC which has all states recurrent nonnull and which is irreducible and homogenous, the* limiting *or* steady-state probability distribution $\{\pi_j\}$ *is given by [15]:*

$$\pi_j = \lim_{t \to \infty} \mathbb{P}(\mathcal{X}(t) = j \mid \mathcal{X}(0) = i)$$

Theorem 2.2 *For an finite, irreducible and homogenous CTMC, the steady-state probabilities $\{\pi_j\}$ always exist and are independent of the initial state distribution. They are uniquely given by the solution of the equations:*

$$-q_{jj}\pi_j + \sum_{k \neq j} q_{kj}\pi_k = 0 \ \textit{ subject to } \ \sum_i \pi_i = 1$$

Again, this can be expressed in matrix vector form (in terms of the vector $\boldsymbol{\pi}$ with elements $\{\pi_1, \pi_2, \ldots, \pi_N\}$ and the matrix $\mathbf{Q}$ defined above) as:

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$$

A CTMC also has an embedded discrete-time Markov chain (EMC) which describes the behaviour of the chain at state-transition instants, that is to say the probability that the next state is j given that the current state is i. The EMC of a CTMC has a one-step $N \times N$ transition matrix $\mathbf{P}$ where $p_{ij} = q_{ij}/-q_{ii}$ for $i \neq j$ and $p_{ij} = 0$ for $i = j$.

2.1.3 Semi-Markov Processes

Semi-Markov Processes (SMPs) are an extension of Markov processes which allow for generally distributed sojourn times. Although the memoryless property no longer holds for state sojourn times, at transition instants SMPs still behave in the same way as Markov processes (that is to say, the choice of the next state is based only on the current state) and so share some of their analytical tractability.

Consider a Markov renewal process $\{(\chi_n, T_n) : n \geq 0\}$ where T_n is the time of the nth transition ($T_0 = 0$) and $\chi_n \in \mathcal{S}$ is the state at the nth transition. Let the kernel of this process be:

$$R(n,i,j,t) = \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i)$$

for $i, j \in \mathcal{S}$. The continuous time semi-Markov process, $\{Z(t), t \geq 0\}$, defined by the kernel R, is related to the Markov renewal process by:

$$Z(t) = \chi_{N(t)}$$

where $N(t) = \max\{n : T_n \leq t\}$, i.e. the number of state transitions that have taken place by time t. Thus $Z(t)$ represents the state of the system at time t. We consider only time-homogenous SMPs in which $R(n,i,j,t)$ is independent of n:

$$\begin{aligned} R(i,j,t) &= \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i) \quad \text{for any } n \geq 0 \\ &= p_{ij} H_{ij}(t) \end{aligned}$$

where $p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i)$ is the state transition probability between states i and j and $H_{ij}(t) = \mathbb{P}(T_{n+1} - T_n \leq t \mid \chi_{n+1} = j, \chi_n = i)$, is the sojourn time distribution in state i when the next state is j. An SMP can therefore be characterised by two matrices $\mathbf{P}$ and $\mathbf{H}$ with elements p_{ij} and H_{ij} respectively.

Semi-Markov processes can be analysed for steady-state performance metrics in the same manner as DTMCs and CTMCs. To do this, we need to know the steady-state probabilities of the SMP's EMC and the average time spent in each state. The first of these can be calculated by solving $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$, as in the case of the DTMC. The average time in state i, $E[\tau_i]$, is the weighted sum of the averages of the sojourn time in the state i when going to state j, $E[\tau_{ij}]$, for all successor states j of i, i.e.:

$$E[\tau_i] = \sum_j p_{ij} E[\tau_{ij}]$$

The steady-state probability of being in state i of the SMP is then [15]:

$$\phi_i = \frac{\pi_i E[\tau_i]}{\sum_{m=1}^{N} \pi_m E[\tau_m]} \tag{2.4}$$

That is, the probability of finding the SMP in state i is the probability of its EMC being in state i multiplied by the average amount of time the SMP spends in state i, normalised over the mean total time spent in all of the states of the SMP.

2.1.4 Classical Iterative Techniques for Steady-state Analysis

There are a number of well-known iterative techniques for computing steady-state probabilities in Markov chains which we will outline very briefly here. These techniques are used for solving linear systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$; in the case of CTMC analysis, $\mathbf{A} = \mathbf{Q}^T$, $\mathbf{x} = \boldsymbol{\pi}^T$ and $\mathbf{b} = \mathbf{0}$ (a vector with all elements being 0), where the superscript T denotes the transpose operator.

Jacobi's Method is the simplest iterative solution technique:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)})$$

where $k \geq 0$ denotes the iteration number and $x^{(0)}$ is an initial guess for the solution vector.

Gauss-Seidel improves on Jacobi by using new x_i iterates as soon as they are computed:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}}{a_{ii}}$$

Successive Over-Relaxation (SOR) accelerates the convergence of the Gauss-Seidel technique by computing iterates as a weighted average of the previous iterate and its newly computed value:

$$x_i^{(k+1)} = \omega \tilde{x}_i^{(k+1)} + (1-\omega) x_i^{(k)}$$

where $\tilde{x}_i^{(k+1)}$ is the ith element of the newly computed solution vector (calculated using the Gauss-Seidel technique), $x_i^{(k)}$ is the ith element of the solution vector at the end of the previous iteration and $0 < \omega < 2$.

2.2 High-level Modelling Formalisms

Specifying every state and state-transition in the state-space of even a moderately complex Markov or semi-Markov chain with perhaps hundreds of states is infeasible. Instead, high-level formalisms can be employed to describe models of systems succinctly; from these the underlying stochastic processes can be extracted automatically and mapped onto a Markov or semi-Markov chain. We describe three such formalisms: stochastic Petri nets, stochastic process algebras and queueing networks.

2.2.1 Petri Nets

Petri nets were invented by Carl Adam Petri in 1962 as a simple graphical formalism for describing and reasoning about concurrent systems [15]. They have been used to model a variety of such systems, including communication protocols, parallel programs, multiprocessor memory caches and distributed databases [88]. Petri initially described Place-Transition nets but numerous other classes of nets have since been defined to allow more for sophisticated reasoning.

The simplest form of Petri net, a Place-Transition net, is formally defined as [15]:

Definition 2.8

A Place-Transition net is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ *where:*

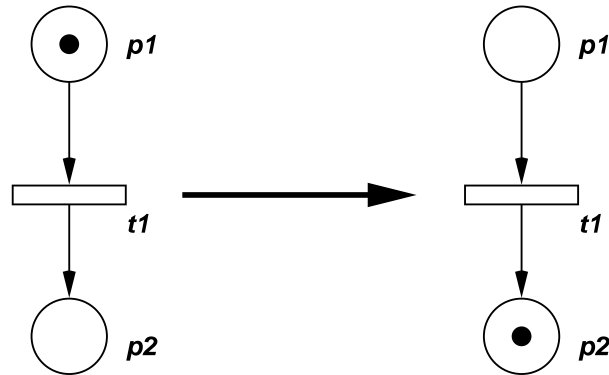

Fig. 2.1. An example Place-Transition net (left) and the result of firing transition t_1 (right).

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places.*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions.*
- $P \cap T = \emptyset$.
- $I^-, I^+ : P \times T \to \mathbb{N}_0$ *are the backward and forward incidence functions, respectively. If* $I^-(p,t) > 0$*, an arc leads from place* p *to transition* t*, and if* $I^+(p,t) > 0$ *then an arc leads from transition* t *to place* p*.*
- $M_0 : P \to \mathbb{N}_0$ *is the initial marking defining the initial number of tokens on every place.*

A marking is a vector of integers representing the number of tokens on each place in a Petri net. The set of all markings that are reachable from the initial marking M_0 is known as the *state-space* or *reachability set* of the Petri net, and is denoted by $R(M_0)$. The connections between markings in the reachability set form the *reachability graph*. If the firing of a transition that is enabled in marking M_i results in marking M_j, then the reachability graph contains a directed arc from marking M_i to marking M_j. Place-Transition nets can be used to reason about qualitative measures such as correctness; for example, if it is possible to reach a marking in which no transitions are enabled then the system can *deadlock*.

The dynamic behaviour of Petri nets centres around the enabling and firing of transitions, which causes tokens to be created and destroyed on places. A transition is

enabled when there are one or more tokens on each of its input places. When that transition fires one or more tokens are destroyed on each of these input places and one or more tokens are created on each of the transition's output places. The exact number of tokens required to enable a transition and the number of tokens created and destroyed by its firing are specified by the backwards and forwards incidence functions respectively.

In the graphical representation of Petri nets, places are represented as circles, transitions as rectangles and tokens as small, filled circles on the places. Consider the simple Place-Transition net on the left in Fig. 2.1. Here, place p_1 is an input place to transition t_1, while place p_2 is an output place of t_1; note that t_1 is enabled as there is a token on p_1. When t_1 fires, the token is destroyed on p_1 and another is created on p_2 – the marking of the net after this occurs is shown on the right of Fig. 2.1.

The formal definition of the enabling and firing rules is [15]:

Definition 2.9 *For a Place-Transition net $PN = (P, T, I^-, I^+, M_0)$:*

- *The* marking *of the net is a function $M : P \mapsto \mathbb{N}_0$, where $M(p)$ denotes the number of tokens on place p.*
- *A transition $t \in T$ is* enabled *in marking M, denoted by $M[t >$, if and only if $M(p) \geq I^-(p,t),\ \forall p \in P$.*
- *A transition $t \in T$ which is enabled in marking M may fire, resulting in a new marking M' such that*

$$M'(p) = M(p) - I^-(p,t) + I^+(p,t),\ \forall p \in P$$

This is denoted $M[t > M'$ and M' is said to be directly reachable *from M, written $M \rightarrow M'$.*

2.2.2 Stochastic Petri Nets

Stochastic Petri Nets are timed extensions of Place-Transition nets where a random exponentially-distributed firing delay is associated with each transition. This means

that the underlying reachability graph is isomorphic to a CTMC. This permits the analysis of models for quantitative performance measures through the analysis of the CTMC, for example the probability of being in a certain set of markings either at equilibrium (steady-state) or at some point in time (transient analysis), or the time to transit between two sets of markings (passage times).

Formally, an SPN is defined as [15]:

Definition 2.10

A Stochastic Petri Net is a tuple $SPN = (PN, \Lambda)$ where:

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.*
- $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ *is the set of, possibly marking dependent, transition rates. Transition t_i has associated rate λ_i.*

When n transitions $t_1, \ldots, t_n$ with corresponding rates $\lambda_1, \ldots, \lambda_n$ are enabled in a given marking, the probability that t_i fires is given by $\lambda_i / \sum_{k=1}^{n} \lambda_k$ [15]. The sojourn time in that marking is then an exponentially-distributed random number with the sum of the rates of the enabled transitions as its parameter, i.e. the rate out of state i is $\mu_i = \sum_{k=1}^{n} \lambda_k$. This is because multiple simultaneously-enabled transitions can be thought of as "racing", and so the one which fires is the one which finishes first – its exponentially-distributed delay must be the lowest of all the delays of the enabled transitions. It is straightforward to show that the minimum of two independent exponentially-distributed random numbers with parameters λ_1 and λ_2 is itself an exponentially-distributed random number with rate parameter $(\lambda_1 + \lambda_2)$ [15].

We further define p_{ij} to be the probability that j is the next marking entered after marking i, μ_i^{-1} to be the mean sojourn time in marking i and $q_{ij} = \mu_i p_{ij}$; i.e. q_{ij} is the instantaneous transition rate into marking j from marking i.

2.2.3 Generalised Stochastic Petri Nets

Generalised Stochastic Petri Nets [5] are timed extensions of Place-Transition nets with two types of transitions: *immediate* transitions and *timed* transitions. Once en-

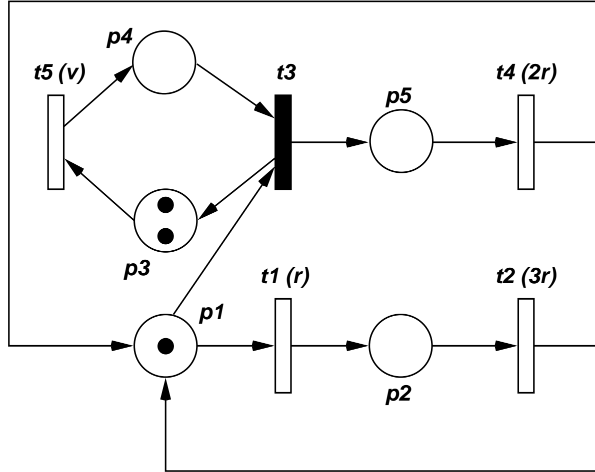

Fig. 2.2. An example Generalised Stochastic Petri Net (GSPN) [48].

abled, immediate transitions fire in zero time, while timed transitions fire after an exponentially-distributed firing delay. Firing of immediate transitions has priority over the firing of timed transitions. The formal definition of a GSPN is as follows [15]:

Definition 2.11

A Generalised Stochastic Petri Net is a 4-tuple $GSPN = (PN, T_1, T_2, C)$ *where:*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
- $T_2 \subset T$ *denotes the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$*,* $T = T_1 \cup T_2$
- $C = (c_1, \ldots, c_{|T|})$ *is an array whose entry* c_i *is either* [†]
  - *a (possibly marking dependent)* **rate** $\in \mathbb{R}^+$ *of an exponential distribution specifying the firing delay, when transition* t_i *is a timed transition (*$t_i \in T_1$*) or*
  - *a (possibly marking dependent)* **weight** $\in \mathbb{R}^+$ *specifying the relative firing frequency, when transition* t_i *is an immediate transition, (*$t_i \in T_2$*)*

[†]Note that we have altered the notation from [15] slightly to avoid confusion between the GSPN probabilistic weight vector, the entries of which depend on the type of transition with which they are associated, and the SM-SPN transition weight function defined below.

The reachability graph of a GSPN contains two types of marking. A *vanishing* marking is one in which an immediate transition is enabled and the sojourn time in such markings is zero. A *tangible* marking is one which enables only timed transitions and the sojourn time in such markings is exponentially distributed with the sum of the rates out of the marking as its parameter (as for SPNs). We denote the set of reachable vanishing markings by $\mathcal{V}$ and the set of reachable tangible markings by $\mathcal{T}$.

In the case where only timed transitions are enabled, firing is conducted in exactly the same manner as for SPNs. When n immediate transitions $t_1, \ldots, t_n$ with weights $c_1, \ldots, c_n$ are enabled, the probability of t_i firing is given by $c_i / \sum_{k=1}^{n} c_k$ [88]. As with SPNs, we define p_{ij} to be the probability that j is the next marking entered after marking i, and, for $i \in \mathcal{T}$, μ_i^{-1} to be the mean sojourn time in marking i and $q_{ij} = \mu_i p_{ij}$; i.e. q_{ij} is the instantaneous transition rate into marking j from marking i.

The stochastic process described by a GSPN's reachability graph is Markovian if $\mathcal{V} = \emptyset$ and semi-Markovian otherwise. It is possible, however, to reduce the reachability graph of a GSPN containing vanishing states to one which is Markovian by using vanishing-state elimination techniques [40, 87].

An example GSPN is shown in Fig. 2.2. The main difference to note over the graphical representation of SPNs is that we are required to distinguish between immediate and timed transitions. Immediate transitions are therefore drawn as filled rectangles (so transition t_3 is the only immediate transitions in the GSPN of Fig. 2.2) while timed transitions are empty rectangles.

2.2.4 Semi-Markov Stochastic Petri Nets

As described above, SPNs and GSPNs (with vanishing states eliminated) have underlying reachability graphs which are isomorphic to CTMCs. It is also possible, however, to generate a semi-Markov process from a stochastic Petri net which does not have exponentially-distributed firing delays on its transitions. Indeed, since 1984 there have been a number of attempts to define non-Markovian stochastic Petri nets [55, 60, 63, 64, 96, 109]. ESPNs [55] were the first proposal for a non-Markovian stochastic

Petri net formalism. Here, general distributions are allowed on transition firings and structural restrictions are imposed on the Petri net to ensure that either a Markov or semi-Markov process is derived. Classes of transitions help to define the structural restrictions: a transition is *exclusive* if, whenever it is enabled, no other transition is enabled; a transition is *competitive* if it is non-exclusive and its firing both interrupts and disables all other enabled transitions; a transition is *concurrent* if it is non-exclusive and its firing does not disable all other enabled transitions. It is established that an SMP can only be generated from an ESPN if transitions with general (GEN) firing-time distributions are constrained to be exclusive. Markovian transitions (with exponentially-distributed firing times), on the other hand, can be both concurrently and competitively enabled.

When more than one GEN transition is enabled [96, 109], then issues of *scheduling policies* for residual pre-empted transition times need to be considered. In [60] three main scheduling policies for pre-empted processes in Markov Regenerative SPNs are presented:

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.

The motivation behind SM-SPNs (developed as part of joint work with Jeremy Bradley and others [24, 25]) is as a specification formalism and higher-level abstraction for semi-Markov models. It is important to note that SM-SPNs do not try to tackle the issue of concurrently enabled GEN transitions in the most general case. If more that

one GEN transition is enabled then a probabilistic choice is used to determine which will be fired. Pre-empted GEN transition use a *prd* schedule if they later become re-enabled. This approach is correctly described in [109] as not being a solution to the more complex issue of properly concurrently enabled GEN transitions, but is merely a way of specifying a different type of model – a semi-Markov model where GEN transitions are essentially forced to be exclusive. Where concurrently enabled GEN transitions do not occur then proper concurrent and competitive transition behaviour is catered for with full *prs* scheduling for pre-empted transitions.

Semi-Markov stochastic Petri nets [24, 25] are extensions of GSPNs [5] which support arbitrary marking-dependent holding-time distributions and generate an underlying semi-Markov process rather than a Markov process. An SM-SPN is defined formally as follows:

Definition 2.12

An SM-SPN is a 4-tuple, $(PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$, where:

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.* P *is the set of places,* T *is the set of transitions,* $I^{+/-}$ *are the forward and backward incidence functions describing the connections between places and transitions and* M_0 *is the initial marking.*

- $\mathcal{P} : T \times \mathcal{M} \to \mathbb{N}_0$*, denoted* $p_t(m)$*, is a marking-dependent priority function for a transition.*

- $\mathcal{W} : T \times \mathcal{M} \to \mathbb{R}^+$*, denoted* $w_t(m)$*, is a marking-dependent weight function for a transition, to allow implementation of probabilistic choice.*

- $\mathcal{D} : T \times \mathcal{M} \to (\mathbb{R}^+ \to [0, 1])$*, denoted* $d_t(m)$*, is a marking-dependent cumulative distribution function for the firing-time of a transition.*

In the above $\mathcal{M}$ is the set of all markings for a given net. Further, we define the following net-enabling functions:

Definition 2.13

- $\mathcal{E}_N : \mathcal{M} \rightarrow P(T)$*, a function that specifies net-enabled transitions from a given marking.*

- $\mathcal{E}_P : \mathcal{M} \rightarrow P(T)$*, a function that specifies priority-enabled transitions from a given marking.*

The net-enabling function $\mathcal{E}_N$ is defined in the usual way for standard Petri nets: if all preceding places have occupying tokens then a transition is net-enabled. The more stringent priority-enabling function $\mathcal{E}_P(m)$ is defined for a given marking m which selects only those net-enabled transitions that have the highest priority, i.e.:

$$\mathcal{E}_P(m) = \{t \in \mathcal{E}_N(m) : p_t(m) = \max\{p_{t'}(m) : t' \in \mathcal{E}_N(m)\}\}$$

For a given priority-enabled transition, $t \in \mathcal{E}_P(m)$, the probability that it will be the one that actually fires (after a delay sampled from its firing distribution $d_t(m)$) is a probabilistic choice based on the relative weights of all enabled transitions:

$$\mathbb{P}(t \in \mathcal{E}_P(m) \text{ fires}) = \frac{w_t(m)}{\sum_{t' \in \mathcal{E}_P(m)} w_{t'}(m)}$$

Note that the choice of which priority-enabled transition is fired in any given marking is made by a probabilistic selection based on transition weights, and is not a race condition based on finding the minimum of samples extracted from firing time distributions. This mechanism enables the underlying reachability graph of the SM-SPN to be mapped directly onto a semi-Markov chain.

To illustrate this enabling and firing strategy, Fig. 2.3 shows an enabled pair of GEN transitions in an SM-SPN. Transition t_1 has a weight of 1.5, a priority of 1 and a $gamma(1.2, 2.3)$ firing distribution, while t_2 has a weight of 1.0, a priority of 1 and a $det(0.01)$ firing distribution. Graphically, each transition is annotated with a 4-tuple specifying the transition name, weight, priority and Laplace transform of its firing time distribution. The weights are used to select which GEN transition will fire: in this case t_1 will be selected to fire with probability $1.5/(1.0 + 1.5) = 0.6$ and p_2 with probability 0.4. After a delay sampled from the selected transition's firing-time distribution, the probabilistic selection takes place again (for the remaining token on p_1); this is followed by another sampled delay.

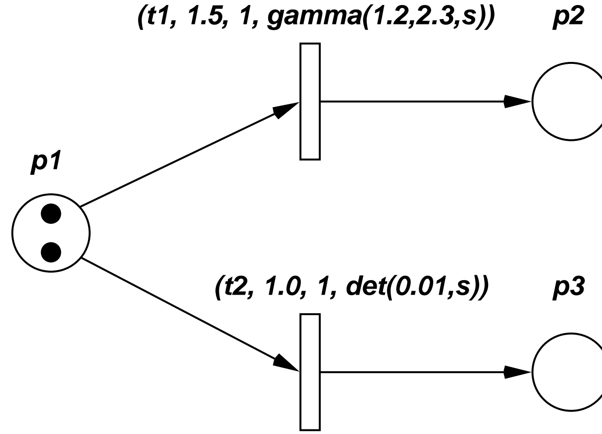

Fig. 2.3. An example Semi-Markov Stochastic Petri Net (SM-SPN) [25].

Expressing SPNs and GSPNs as SM-SPNs

The marking-dependence of the weights and distributions allows the translation of SPNs and GSPNs into the SM-SPN paradigm in a straightforward manner. An SPN can be specified in the SM-SPN formalism in the following way:

- $p_t(m) = 0 \quad \forall t, m$
- $w_t(m) = \lambda_t \quad \forall t, m$
- $d_t(m) = F(r)$ where $F(r) = 1 - \exp(-\lambda^{\Sigma} r) \quad \forall t, m$

where $\lambda^{\Sigma} = \sum_{t \in \mathcal{E}_N(m)} \lambda_t$, the sum of all the rates of the enabled transitions in marking m.

For GSPNs, the translation to an SM-SPN necessarily distinguishes between timed transitions ($t \in T_1$) and immediate transitions ($t \in T_2$):

- $p_t(m) = \begin{cases} 0 & : t \in T_1 \\ 1 & : t \in T_2 \end{cases}$
- $w_t(m) = c_t \quad$ for all t
- $d_t(m) = \begin{cases} F(r) & : t \in T_1 \\ 1 & : t \in T_2 \end{cases}$

where $F(r) = 1 - \exp(-\lambda^{\Sigma} r)$ and $\lambda^{\Sigma} = \sum_{t \in \mathcal{E}_P(m)} \lambda_t$. This gives us a meaningful combined exponential rate, λ^{Σ}, if only timed transitions are priority enabled.

2.2.5 Stochastic Process Algebras

Process algebras are another formalism for describing concurrent systems but, unlike Petri nets, they are not a graphical formalism. Instead, a model is formed of a textual description (the format of which is dependent on the process algebra being used) of a group of *processes* or *components*. These can perform *actions* either individually or by synchronising with each other. This component-oriented structure permits *compositionality* in model design, so a description of a system can be constructed from the individual descriptions of its components. Like Place-Transition nets, process algebras can be used to reason about the correctness of systems, for example whether or not they can deadlock. Examples of process algebras are Calculus of Communicating Systems (CCS) [100, 101] and Communicating Sequential Processes (CSP) [76, 77].

Stochastic Process Algebras (SPAs) introduce the notion of time into process algebras, usually by associating random durations with actions. This allows for the analysis of quantitative performance measures to be undertaken (in the same manner in which SPNs extend Place-Transition nets). As an example of a Markovian SPA we briefly describe Performance Enhanced Process Algebra (PEPA) [75].

Definition 2.14 *The syntax of a PEPA component, P, is given by:*

$$P \quad ::= \quad (a, \lambda).P \;\Big|\; P + P \;\Big|\; P \underset{S}{\bowtie} P \;\Big|\; P/L \;\Big|\; A$$

where:

- $(a, \lambda).P$ *is the* prefix operator. *A process performs an* action*, a, and then becomes a new process, P. For* active *actions, the time taken to perform a is an exponentially distributed random variable with parameter λ. The rate parameter may also take a $\top$-value, which makes the action* passive *in a cooperation (see below).*

- $P_1 + P_2$ *is the* choice operator. *A race is entered into between components* P_1 *and* P_2 *to determine which will complete its action first. If* P_1 *evolves first then any behaviour of* P_2 *is lost and vice versa.*

- $P_1 \underset{S}{\bowtie} P_2$ *is the* cooperation operator. P_1 *and* P_2 *run in parallel and synchronise over the set of actions in the set* S*. If* P_1 *is to evolve with an action* $a \in S$*, then it must wait for* P_2 *to be in a position to produce an* a*-action before it can proceed, and vice versa. In an* active *cooperation, the two components then jointly perform an* a*-action with a rate based on that of the slower of the two components. In a* passive *cooperation, where one of the processes will be performing action* a *with parameter* $\top$*, the cooperation proceeds at the rate of the active action only.*

- P/L *is the* hiding operator *where actions in the set* L *which can be performed by component* P *are rewritten as silent* τ *actions (although they maintain their original delay parameters). The actions in* L *can no longer be used to cooperate with other components.*

- A *is a* constant label *and allows recursive definitions to be constructed.*

Fig. 2.4 shows an example PEPA model which demonstrates the use of the choice and cooperation operators. There are two machines, $Machine_A$ and $Machine_B$, each with an associated monitoring alarm, $Alarm_A$ and $Alarm_B$. Both machines share the behaviour that when they are paused (when they are behaving as $Machine_x1$) they can perform a $start$ action to begin running. Once running, they can both $pause$, but $Machine_A$ can also $fail$ and must then $recover$. The two monitoring processes, $Alarm_A$ and $Alarm_B$, will raise an $alert$ when they observe a run action in their corresponding machine. This behaviour is enforced by the composition of the system, as the alarms cooperate with the machines over the run action. Finally, the system as whole will only display an $alert$ when both the $Machine_A/Alarm_A$ and $Machine_B/Alarm_B$ pairs are displaying an $alert$.

As all delays in PEPA models are exponentially distributed, a CTMC can be generated from a model's derivation graph (which is analogous to the reachability graph of a Petri

$$
\begin{aligned}
Machine_A1 &\stackrel{\text{def}}{=} (start, r_1).Machine_A2 + (pause, r_2).Machine_A3 \\
Machine_A2 &\stackrel{\text{def}}{=} (run, r_3).Machine_A1 + (fail, r_4).Machine_A3 \\
Machine_A3 &\stackrel{\text{def}}{=} (recover, r_1).Machine_A1 \\
Alarm_A &\stackrel{\text{def}}{=} (run, \top).(alert, r_5).Alarm_A
\end{aligned}
$$

$$
\begin{aligned}
Machine_B1 &\stackrel{\text{def}}{=} (start, r_1).Machine_B2 + (pause, r_2).Machine_B1 \\
Machine_B2 &\stackrel{\text{def}}{=} (run, r_3).Machine_B1 \\
Alarm_B &\stackrel{\text{def}}{=} (run, \top).(alert, r_5).Alarm_B
\end{aligned}
$$

$$
System \stackrel{\text{def}}{=} (Alarm_A \underset{run}{\bowtie} Machine_A1) \underset{alert}{\bowtie} (Alarm_B \underset{run}{\bowtie} Machine_B1)
$$

Fig. 2.4. An example PEPA model.

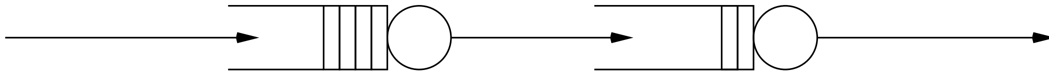

Fig. 2.5. An example open queueing network.

net). This Markov chain can then be analysed for steady-state or other performance measures in exactly the same fashion as a CTMC generated from an SPN or a GSPN.

2.2.6 Queueing Networks

The final high-level modelling formalism we consider is queueing networks. Treatment of them can be found in numerous works, for example [15, 71, 103]. Queueing networks are built from three basic components:

- *Servers* may have one or more queues connected to them. The time taken for a server to process a customer is a random variable which may be drawn from a number of distributions including the exponential. Servers also have a *schedul-*

ing strategy, which specifies the order in which queued customers are served – a common example is First-Come First-Served (FCFS).

- *Queues* store customers in the order in which they arrive at a server. Queues may have a fixed capacity or have the ability to store an infinite number of customers.
- *Customers* move between queues and are processed by servers. They may be divided into different *classes*, which will affect how they are routed between queues and how they are served when they arrive at them (for example, different classes may be assigned different priority levels and the arrival of a high-priority customer at a queue may pre-empt the service of a lower priority customer).

Additionally, when departures from one server can become arrivals at one of several destination queues it is necessary to specify *routing probabilities*, which are the probabilities that a customer which leaves that server will be routed to each of the possible destinations. Different classes of customers can have different routing probabilities.

Queueing networks can be classed as either *open* or *closed*. When the population of customers in the network is fixed (i.e. it is not possible for customers to leave the network or for new customers to join it), the network is closed. Otherwise, it is said to be open.

Queues in a queueing network are often described using the Kendall notation:

$$A/B/m/K/Z/S$$

in which the components identify the arrival process, the service distribution, the number of servers, queue capacity, customer population (in a closed network) and scheduling strategy respectively. If not specified, the queue capacity is assumed to be infinite and the scheduling strategy is assumed to be FCFS. A commonly encountered single-server queue is the $M/M/1$ queue, in which arrivals and services are exponentially-distributed.

For closed queueing networks with Markovian service times, it is possible to generate a reachability graph which is isomorphic to a CTMC (a state in the CTMC being described by the number of customers at each queue). This can then be analysed for

steady-state, transient and passage time quantities in exactly the same manner as a CTMC derived from a Petri net or SPA model.

2.3 Laplace Transforms

The Laplace transform is an integral transform which is widely used in the solution of differential equations arising from physical problems such as heat conduction, movement of bodies and so forth. Such problems are typically hard to solve in (real-valued) t-space, but can be transformed into (complex-valued) s-space using the Laplace transform and solved more easily before being transformed back into t-space. This approach can also be adopted for the passage time analysis of Markov and semi-Markov chains.

When the Laplace transform $f^*(s)$ of a real-valued function $f(t)$ exists, it is unique and is given by:

$$f^*(s) = \int_0^\infty e^{-st} f(t)\, dt \tag{2.5}$$

where s is a complex number.

For the Laplace transform of a function $f(t)$ to exist, $f(t)$ must be of *exponential order*. Examples of functions which meet this restriction are polynomial or exponential (those of the form e^{kt}) functions and bounded functions. Also included are those functions with a finite number of finite discontinuities. Examples of functions which do not fall within this category are those which have singularities (e.g. $ln(x)$), those whose growth rates are faster than exponential (e.g. e^{x^2}) or those with an infinite number of finite discontinuities (e.g. $f(x) = 1$ if x is rational and 0 otherwise).

For the purposes of this thesis, the functions considered are all probability density functions and so are sufficiently 'well-behaved' (i.e. they have area underneath them which integrates to 1) that their Laplace transforms will always exist. Note, however, that they may not always exist in closed form (e.g. as is the case for the Weibull distribution).

2.3.1 Properties

One example of where Laplace transforms are useful is the calculation of the convolution of two functions, an operation which is of particular importance in passage time analysis. The calculation of the probability density function of a passage time between two states is achieved by convolving the probability density functions of the sojourn times of the states along all the paths between the source and target states. The convolution of two functions $f(t)$ and $g(t)$ denoted $f(t) * g(t)$ is given by:

$$f(t) * g(t) = \int_0^t f(\tau)g(t-\tau)\, d\tau \tag{2.6}$$

The convolution of n functions requires the evaluation of an $(n-1)$ dimensional integral. To perform such a calculation for large values of n (perhaps in the millions) would be impractical. Instead, we exploit the convolution property of Laplace transforms, which states that the Laplace transform of the convolution of two functions is the product of the functions' individual Laplace transforms. Once the Laplace transform of the convolution has been calculated in terms of the complex parameter s, it is possible to retrieve the convolution in terms of the real-valued parameter t using a process known as *Laplace transform inversion*.

Theorem 2.3 Convolution: *The Laplace transform of the convolution of two functions $f(t)$ and $g(t)$, denoted $L\{f(t) * g(t)\}$, is the product of the Laplace transforms of the two functions, i.e.:*

$$L\{f(t) * g(t)\} = f^*(s)g^*(s)$$

The proof for Theorem 2.3 is well known and we will outline it briefly here in the manner presented in [56]. Let $f(t)$ and $g(t)$ be two functions whose Laplace transforms exist and are denoted $f^*(s)$ and $g^*(s)$ respectively. We write the Laplace transform of the convolution of these two functions as $L\{f(t) * g(t)\}$, and wish to prove:

$$L\{f(t) * g(t)\} = f^*(s)\, g^*(s)$$

Starting from the definition of the Laplace transform in Eq. 2.5 and of the convolution of $f(t)$ and $g(t)$ in Eq. 2.6, we can write:

$$L\{f(t) * g(t)\} = \int_0^\infty e^{-st} \int_0^t f(\tau)g(t-\tau)\, d\tau\, dt$$

We can rewrite this double integral as:

$$L\{f(t) * g(t)\} = \int_0^\infty \int_0^t e^{-st} f(\tau)g(t-\tau)\, d\tau\, dt$$

and then change the order and limits of integration:

$$\begin{aligned} L\{f(t) * g(t)\} &= \int_0^\infty \int_\tau^\infty e^{-st} f(\tau)g(t-\tau)\, dt\, d\tau \\ &= \int_0^\infty f(\tau) \left(\int_\tau^\infty e^{-st} g(t-\tau)\, dt \right) d\tau \end{aligned} \tag{2.7}$$

Substituting the variable $u = t - \tau$ into the inner integral of Eq. 2.7 gives:

$$\begin{aligned} \int_\tau^\infty e^{-st}\, g(t-\tau)\, dt &= \int_0^\infty e^{-s(u+\tau)}\, g(u)\, du \\ &= e^{-s\tau} \int_0^\infty e^{-su}\, g(u)\, du \\ &= e^{-s\tau} g^*(s) \end{aligned}$$

and substituting this back into Eq. 2.7 yields:

$$\begin{aligned} L\{f(t) * g(t)\} &= \int_0^\infty f(\tau)\, e^{-s\tau}\, g^*(s)\, d\tau \\ &= g^*(s) \int_0^\infty e^{-s\tau}\, f(\tau)\, d\tau \\ &= g^*(s)\, f^*(s) \\ &= f^*(s)\, g^*(s) \end{aligned}$$

Thus the convolution of the functions $f(t)$ and $g(t)$ can be obtained by inverting the Laplace transform of the convolution. The next section describes how this inversion can be achieved using numerical methods.

Another property of Laplace transforms is *linearity* [56]:

Theorem 2.4 Linearity: *If $f(t)$ and $g(t)$ are functions whose Laplace transforms exist, then:*

$$L\{af(t) + bg(t)\} = aL\{f(t)\} + bL\{g(t)\}$$

Proof:

$$\begin{aligned} L\{af(t)+bg(t)\} &= \int_0^\infty (af(t)+bg(t))e^{-st}\,dt \\ &= \int_0^\infty af(t)e^{-st}+bg(t)e^{-st}\,dt \\ &= a\int_0^\infty f(t)e^{-st}dt + b\int_0^\infty g(t)e^{-st}dt \\ &= aL\{f(t)\}+bL\{g(t)\} \end{aligned}$$

A final property of Laplace transforms which is particularly useful in the context of this work is that the Laplace transform of a cumulative distribution function can be calculated from the Laplace transform of the corresponding probability density function by dividing it by s. This corresponds to integration of $f(t)$ in t-space.

Theorem 2.5 Integration: *If $f(t)$ is a probability density function and $F(t)$ is the corresponding cumulative distribution function, $\int_0^\infty f(t)\ dt = F(t)$. The Laplace transform of $F(t)$ can be calculated from the Laplace transform of $f(t)$ by dividing $L\{f(t)\}$ by s:*

$$L\{F(t)\} = L\{f(t)\}/s$$

Proof:

Recalling that the integration by parts of a definite interval $\int_a^b u\,dv = [uv]_a^b - \int_a^b v\,du$:

$$\begin{aligned} \int_0^\infty e^{-st}f(t)\,dt &= \left[e^{-st}F(t)\right]_0^\infty - \int_0^\infty -se^{-st}F(t)\,dt \\ &= s\int_0^\infty e^{-st}F(t)\,dt \end{aligned}$$

So $L\{f(t)\} = sL\{F(t)\}$ or $L\{F(t)\} = L\{f(t)\}/s$.

2.3.2 Laplace Transform Inversion

As the Laplace transform of a function is unique it is possible to recover the function $f(t)$ from its Laplace transform $f^*(s)$. This process is called Laplace transform inversion. The inverse of the Laplace transform $f^*(s)$ of a function $f(t)$ (which we denote $L^{-1}\{f^*(s)\}$) is the function $f(t)$ itself:

$$L^{-1}\{f^*(s)\} = f(t) = \frac{1}{2\pi i}\int_{a-i\infty}^{a+i\infty} e^{st}f^*(s)\,ds \tag{2.8}$$

where a is a real number which lies to the right of all the singularities of $f^*(s)$. This is also known as the *Bromwich contour inversion integral*.

Like Laplace transforms, inverse Laplace transforms display linearity [56]:

$$L^{-1}\{af^*(s) + bg^*(s)\} = aL^{-1}\{f^*(s)\} + bL^{-1}\{g^*(s)\}$$

The work in this thesis centres around the calculation and *numerical* inversion of Laplace transforms. There are a number of numerical Laplace transform inversion algorithms in the literature, for example the Euler technique [3, 4], Talbot's technique [118] and the Laguerre method [2] (also known as Weeks' method [124]). We now summarise the key features of these methods.

Summary of Euler Inversion

It is possible to rewrite Eq. 2.8 such that it is possible to obtain $f(t)$ from $f^*(s)$ by integrating a real-valued function of a real variable rather than requiring contour integration to be performed in complex space. Substituting $s = a + iu$ allows Eq. 2.8 to be rewritten as [1]:

$$f(t) = \frac{1}{2\pi i} \int_{-\infty}^{\infty} e^{(a+iu)t} f^*(a + iu)\, du$$

Making use of the fact that:

$$e^{(b+iu)t} = e^{bt}(\cos ut + i \sin ut)$$

yields [1]:

$$f(t) = \frac{2e^{at}}{\pi} \int_{0}^{\infty} \mathrm{Re}(f^*(a + iu))\ \cos(ut)\, du$$

This integral can be evaluated numerically using the trapezoidal rule. This approximates the integral of a function $f(t)$ over the interval $[a, b]$ as:

$$\int_{a}^{b} f(t)\, dt \approx h \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f(a + kh) \right)$$

where $h = (b - a)/n$. Setting the step-size $h = \pi/2t$ and $a = A/2t$ (where A is a constant that controls the discretisation error and is set to 19.1 in [4]) results in the

alternating series [4, 53]:

$$f(t) \approx \frac{e^{A/2}}{2t}\mathrm{Re}\left(f^*\left(\frac{A}{2t}\right)\right) + \frac{e^{A/2}}{2t}\sum_{k=1}^{\infty}(-1)^k\,\mathrm{Re}\left(f^*\left(\frac{A+2k\pi i}{2t}\right)\right)$$

Euler summation can be employed to accelerate the convergence of this alternating series [115]. That is, we calculate the sum of the first n terms explicitly and use Euler summation to calculate the next m. The mth term after the first n is given by [3]:

$$E(t,m,n) = \sum_{k=0}^{m}\binom{m}{k}2^{-m}\,s_{n+k}(t) \tag{2.9}$$

In Eq. 2.9:

$$s_n(t) = \sum_{k=0}^{n}(-1)^k\,\mathrm{Re}\left(f^*\left(\frac{A+2k\pi i}{2t}\right)\right)$$

An estimate of the truncation error incurred in using Euler summation can be calculated by comparing the magnitudes of the nth and $(n+1)$th terms, i.e. [3]:

$$|E(t,m,n) - E(t,m,n+1)|$$

To give a truncation error of 10^{-8} we set $n = 20$ and $m = 12$.

Summary of Talbot's Method

Talbot's method is very similar to Euler inversion in that it involves the trapezoidal integration of Eq. 2.8 [118]. However, the contour over which the integration is performed is altered from the one in Eq. 2.8 in such a way that the oscillations in the terms of the infinite summation (resulting from the application of the trapezoidal rule) are avoided and therefore no technique is needed to accelerate the convergence of the series. The new contour, however, must enclose all singularities of $f^*(s)$, which means a record must be kept of the locations of these singularities [54]. In the context of the work presented in this thesis this would not be practical: not only would a list of the singularities of all state holding time density function Laplace transforms need to be stored, but it would have to be maintained for the Laplace transforms of the convolutions of these functions as well. For this reason, the Talbot method is not employed in any of the work which follows.

Summary of Laguerre Inversion

The Laguerre method [2] makes use of the Laguerre series representation of $f(t)$:

$$f(t) = \sum_{n=0}^{\infty} q_n l_n(t) \qquad : t \geq 0$$

where the Laguerre polynomials l_n are given by:

$$l_n(t) = \left(\frac{2n-1-t}{n}\right) l_{n-1}(t) - \left(\frac{n-1}{n}\right) l_{n-2}(t)$$

starting with $l_0 = e^{t/2}$ and $l_1 = (1-t)e^{t/2}$, and:

$$q_n = \frac{1}{2\pi r^n} \int_0^{2\pi} Q(re^{iu})e^{-inu}\, du \tag{2.10}$$

where $r = (0.1)^{4/n}$ and $Q(z) = (1-z)^{-1} f^*\big((1+z)/2(1-z)\big)$.

The integral in the calculation of Eq. 2.10 can be approximated numerically using the trapezoidal rule, giving:

$$q_n \approx \frac{1}{2nr^n} \left(Q(r) + (-1)^n Q(-r) + 2\sum_{j=1}^{n-1} (-1)^j \mathrm{Re}\left(Q(re^{\pi ji/n})\right) \right) \tag{2.11}$$

As described in [70], the Laguerre method can be modified by noting that the Laguerre coefficients q_n are independent of t. Since $|l_n(t)| \leq 1$ for all n, the convergence of the Laguerre series depends on the decay rate of q_n as $n \to \infty$ which is in turn determined by the smoothness of $f(t)$ and its derivatives [2]. Slow convergence of the q_n coefficients can often be improved by exponential dampening and scaling using two real parameters σ and b [124]. Here the inversion algorithm is applied to the function

$$f_{\sigma,b}(t) = e^{-\sigma t} f(t/b)$$

with $f(t)$ being recovered as:

$$f(t) = e^{\sigma bt} f_{\sigma,b}(bt).$$

Each q_n coefficient is computed as in Eq. 2.11, using the trapezoidal rule with $2n$ trapezoids. However, if we apply scaling to ensure that q_n has decayed to (almost) zero by term p_0 (say $p_0 = 200$), we can instead make use of a constant number of

$2p_0$ trapezoids when calculating each q_n. This allows us to calculate each q_n with high accuracy while simultaneously providing the opportunity to cache and re-use values of $Q(z)$. Since q_n does not depend on t, and each evaluation of $Q(z)$ involves a single evaluation of $f^*(s)$, we obtain $f(t)$ at an arbitrary number of t-values at the fixed cost of evaluating $Q(z)$ (and hence $f^*(s)$) $2p_0$ times.

Suitable scaling parameters can be automatically determined using the algorithm in Fig. 2.6 [70]. This algorithm is based on the heuristic observations in [2] that increasing b (up to a given limit) can significantly lower the ratio $|q_n|/|q_0|$, and the observation in [70] that excessive values of the damping parameter σ can lead to numerical instability in finite precision arithmetic.

$\sigma = 0.0$
$b = 1$
while $|q_{200}| > 10^{-10}$ **or** $|q_{201}| > 10^{-10}$ **do begin**
    if $\sigma = 0$ **then**
        $\sigma = 0.001$
    else
        $\sigma = 2\sigma$
    if $\sigma > 0.2$ **then begin**
        $b = b + 4$
        if $b > 10$
            exit
        $\sigma = 0$
    end
end

Fig. 2.6. Algorithm for automatically determining scaling parameters [70].

This fixed computational cost for any number of t-points is in contrast to the Euler method, where the number of different s-values at which $f^*(s)$ must be evaluated is

a function of the number of points at which the value of $f(t)$ is required (in fact, it is $(n + m + 1)$ times the number of t-points). It must be noted, however, that the Euler method can be used to invert Laplace transforms which are not sufficiently smooth to permit the modified Laguerre method to be used. This situation typically arises when the original function $f(t)$ has discontinuities (e.g., if $f(t) = det(x)$).

Chapter 3

Passage Times in Markov Models

This chapter first describes the calculation of passage time densities and quantiles in continuous-time Markov chains using the Laplace transform technique presented in [70]. We then present a novel contribution of this thesis, namely the extraction of passage times in systems modelled using GSPNs which can have both tangible or vanishing source and target states [48]. We also describe a second technique for the calculation of passage time densities in Markov models known as uniformization. We present a comparison of the run-time behaviour of the Laplace transform and uniformization techniques and contrast them both with simulation. Extraction of passage times from stochastic process algebra models using uniformization is demonstrated. Finally, we describe a low-cost approximation technique which estimates passage time densities and distributions from their moments [7, 8].

3.1 The Laplace Transform Method for CTMCs

Consider a finite, irreducible CTMC with N states $\{1, 2, \ldots, N\}$ and generator matrix $\mathbf{Q}$ as defined in Section 2.1.2. As $\chi(t)$ denotes the states of the CTMC at time t ($t \geq 0$) and $N(t)$ denotes the number of state transitions which have occurred by time t, the first passage time from a single source marking i into a non-empty set of target

markings $\vec{j}$ is:

$$P_{i\vec{j}}(t) = \inf\{u > 0 : \chi(t+u) \in \vec{j}, N(t+u) > N(t), \chi(t) = i\}$$

When the CTMC is stationary and time-homogenous this quantity is independent of t:

$$P_{i\vec{j}} = \inf\{u > 0 : \chi(u) \in \vec{j}, N(u) > 0, \chi(0) = i\} \tag{3.1}$$

That is, the first time the system enters a state in the set of target states $\vec{j}$, given that the system began in the source state i and at least one state transition has occurred. $P_{i\vec{j}}$ is a random variable with probability density function $f_{i\vec{j}}(t)$ such that:

$$\mathbb{P}(a < P_{i\vec{j}} < b) = \int_a^b f_{i\vec{j}}(t)\, dt \quad (0 \le a < b)$$

In order to determine $f_{i\vec{j}}(t)$ it is necessary to convolve the state holding-time density functions over all possible paths (including cycles) from state i to all of the states in $\vec{j}$.

As described in Section 2.3, the calculation of the convolution of two functions in t-space (cf. Eq. 2.6) can be more easily accomplished by multiplying their Laplace transforms together in s-space and inverting the result. The calculation of $f_{i\vec{j}}(t)$ is therefore achieved by calculating the Laplace transform of the convolution of the state holding times over all paths between i and $\vec{j}$ and then numerically inverting this Laplace transform.

In a CTMC all state sojourn times are exponentially distributed, so the density function of the sojourn time in state i is $\mu_i e^{-\mu_i t}$, where $\mu_i = -q_{ii}$ for $1 \le i \le N$ as defined in Section 2.1.2. The Laplace transform of an exponential density function with rate parameter λ can be calculated from Eq. 2.5:

$$\begin{aligned}
L\{\lambda e^{-\lambda t}\} &= \int_0^\infty e^{-st}(\lambda e^{-\lambda t})\, dt \\
&= \lambda \int_0^\infty e^{-st-\lambda t}\, dt \\
&= \lambda \int_0^\infty e^{-(s+\lambda)t}\, dt \\
&= \lambda \left[\frac{-1}{(s+\lambda)} e^{-(s+\lambda)t} \right]_0^\infty \\
&= \frac{\lambda}{(s+\lambda)}
\end{aligned}$$

Denoting the Laplace transform of the density function $f_{i\vec{j}}(t)$ of the passage time random variable $P_{i\vec{j}}$ as $L_{i\vec{j}}(s)$, we proceed by means of a first-step analysis. That is, to calculate the first passage time from state i into the set of target states $\vec{j}$, we consider moving from state i to its set of direct successor states $\vec{k}$ and thence from states in $\vec{k}$ to states in $\vec{j}$. This can be expressed as the following system of linear equations:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} p_{ik} \left(\frac{-q_{ii}}{s - q_{ii}} \right) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} \left(\frac{-q_{ii}}{s - q_{ii}} \right) \quad (3.2)$$

The first term (i.e. the summation over non-target states $k \notin \vec{j}$) convolves the sojourn time density in state i with the density of the time taken for the system to evolve from state k into a target state $\in \vec{j}$, weighted by the probability that the system transits from state i to state k. The second term (i.e. the summation over target states $k \in \vec{j}$) simply reflects the sojourn time density in state i weighted by the probability that a transition from state i into a target state k occurs.

Given that $p_{ij} = q_{ij}/-q_{ii}$ in the context of a CTMC (cf. Section 2.1.2), Eq. 3.2 can be rewritten as:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} \left(\frac{q_{ik}}{s - q_{ii}} \right) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} \left(\frac{q_{ik}}{s - q_{ii}} \right) \quad (3.3)$$

This set of linear equations can be expressed in matrix–vector form. For example, when $\vec{j} = \{1\}$ we have:

$$\begin{pmatrix} s - q_{11} & -q_{12} & \cdots & -q_{1n} \\ 0 & s - q_{22} & \cdots & -q_{2n} \\ 0 & -q_{32} & \cdots & -q_{3n} \\ 0 & \vdots & \ddots & \vdots \\ 0 & -q_{n2} & \cdots & s - q_{nn} \end{pmatrix} \begin{pmatrix} L_{1\vec{j}}(s) \\ L_{2\vec{j}}(s) \\ L_{3\vec{j}}(s) \\ \vdots \\ L_{n\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} 0 \\ q_{21} \\ q_{31} \\ \vdots \\ q_{n1} \end{pmatrix} \quad (3.4)$$

Our formulation of the passage time quantity in Eq. 3.1 states that we must observe at least one state-transition during the passage. In the case where $i \in \vec{j}$ (as for $L_{1\vec{j}}(s)$ in the above example), we therefore calculate the density of the cycle time to return to state i rather than requiring $L_{i\vec{j}}(s) = 1$.

Given a particular (complex-valued) s, Eq. 3.4 can be solved for $L_{i\vec{j}}(s)$ by standard iterative numerical techniques for the solution of systems of linear equations in $\mathbf{Ax} = \mathbf{b}$

form (cf. Section 2.1.4). In the context of the inversion algorithms described in Section 2.3.2, both Euler and Laguerre can identify in advance at which values of s $L_{i\vec{j}}(s)$ must be calculated in order to perform the numerical inversion. Therefore, if the algorithm requires m different values of $L_{i\vec{j}}(s)$, Eq. 3.4 will need to be solved m times.

The corresponding cumulative distribution function $F_{i\vec{j}}(t)$ of the passage time is obtained by integrating under the density function. As described in Section 2.3, this integration can be achieved in terms of the Laplace transform of the density function by dividing it by s, i.e. $F^*_{i\vec{j}}(s) = L_{i\vec{j}}(s)/s$. In practice, if Eq. 3.4 is solved as part of the inversion process for calculating $f_{i\vec{j}}(t)$, the m values of $L_{i\vec{j}}(s)$ can be retained. Once the numerical inversion algorithm has used them to compute $f_{i\vec{j}}(t)$, these values can be recovered, divided by s and then taken as input by the numerical inversion algorithm again to compute $F_{i\vec{j}}(t)$. Thus, in calculating $f_{i\vec{j}}(t)$, we get $F_{i\vec{j}}(t)$ for little further computational effort.

When there are multiple source markings, denoted by the vector $\vec{i}$, the Laplace transform of the response time density at equilibrium is:

$$L_{\vec{i}\vec{j}}(s) = \sum_{k \in \vec{i}} \alpha_k L_{k\vec{j}}(s)$$

where the weight α_k is the equilibrium probability that the state is $k \in \vec{i}$ at the starting instant of the passage. This instant is the moment of entry into state k; thus α_k is proportional to the equilibrium probability of the state k in the underlying embedded (discrete-time) Markov chain (EMC) of the CTMC with one-step transition matrix $\mathbf{P}$ as defined in Section 2.1.2. That is,

$$\alpha_k = \begin{cases} \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{if } k \in \vec{i} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

where the vector $\boldsymbol{\pi}$ is any non-zero solution to $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$. The row vector with components α_k is denoted by $\boldsymbol{\alpha}$.

3.2 Extension to GSPNs

The analysis described above for Markov chains can be extended to the analysis of the underlying state spaces of GSPNs [48]. The situation is complicated, however, by the existence of two types of state (tangible and vanishing) as described in Section 2.2.3. As we are dealing with Petri nets, the analysis is described in terms of markings rather than states (although the two terms are equivalent – the state of a GSPN is defined by its marking).

In a GSPN, the first passage time from a single source marking i into a non-empty set of target markings $\vec{j}$ is:

$$P_{i\vec{j}} = \inf\{u > 0 : M(u) \in \vec{j}, N(u) > 0, M(0) = i\}$$

where $M(t)$ is the marking of the GSPN at time t and $N(t)$ denotes the number of state transitions which have occurred by time t.

We proceed by means of a first-step analysis as described above for the purely Markovian case. Recalling the definitions of Section 2.2.2, the Laplace transform of the (exponential) sojourn time density function of tangible marking i is $\mu_i/(s + \mu_i)$, but for a vanishing marking the sojourn time is 0 with probability 1, giving a corresponding Laplace transform of 1 for all values of s.[†] We must therefore distinguish between passage times which start in a tangible state and those which begin in a vanishing state:

$$L_{i\vec{j}}(s) = \begin{cases} \sum_{k \notin \vec{j}} \left(\frac{q_{ik}}{s - q_{ii}}\right) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} \left(\frac{q_{ik}}{s - q_{ii}}\right) & \text{if } i \in \mathcal{T} \\ \\ \sum_{k \notin \vec{j}} p_{ik} L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} & \text{if } i \in \mathcal{V} \end{cases} \quad (3.6)$$

Again, this system of linear equations can be expressed in matrix–vector form. For example, when $\vec{j} = \{1\}$, $\mathcal{V} = \{2\}$ and $\mathcal{T} = \{1, 3, \ldots, n\}$ the above equations can be

[†]This also follows as the probability density function of an immediate transition is an impulse function at $x = 0$. The Laplace transform of an impulse function at $x = a$ is e^{-as} [107], which is 1 when $a = 0$.

written as:

$$\begin{pmatrix} s-q_{11} & -q_{12} & \cdots & -q_{1n} \\ 0 & 1 & \cdots & -p_{2n} \\ 0 & -q_{32} & \cdots & -q_{3n} \\ 0 & \vdots & \ddots & \vdots \\ 0 & -q_{n2} & \cdots & s-q_{nn} \end{pmatrix} \begin{pmatrix} L_{1\vec{j}}(s) \\ L_{2\vec{j}}(s) \\ L_{3\vec{j}}(s) \\ \vdots \\ L_{n\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} 0 \\ p_{21} \\ q_{31} \\ \vdots \\ q_{n1} \end{pmatrix}$$

This system of linear equations can then be solved by the same techniques as for the Markov case above.

As described above in Section 3.1, this formulation can easily be generalised to the case where multiple source states are required. This is accomplished by weighting the $L_{k\vec{j}}(s)$ values with the renormalised steady-state probabilities for state $k \in \vec{i}$ from the embedded Markov chain (EMC) defined by the marking of the GSPN at firing instants. Therefore,

$$\alpha_k = \begin{cases} \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{if } k \in \vec{i} \\ 0 & \text{otherwise} \end{cases}$$

where the vector $\boldsymbol{\pi}$ is any non-zero solution to $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$.

Note also that if vanishing states are eliminated from the underlying state space during its generation, the result is a continuous-time Markov chain which can then be analysed for passage times as per Section 3.1. Doing so reduces the size of the state space to be analysed but removes the ability to reason about source or target states which are vanishing.

3.2.1 Example of GSPN Analysis

Fig. 3.1 shows a small contrived GSPN model and Fig. 3.2 its corresponding reachability graph. We illustrate our technique on this GSPN by computing the response time density for the time taken to reach markings where $M(p_2) > 0$ from markings where $M(p_1) > 0$.

In this example, there are three source markings, two of which are vanishing and one of which is tangible. As discussed in Section 3.2, the Laplace transforms of the passage

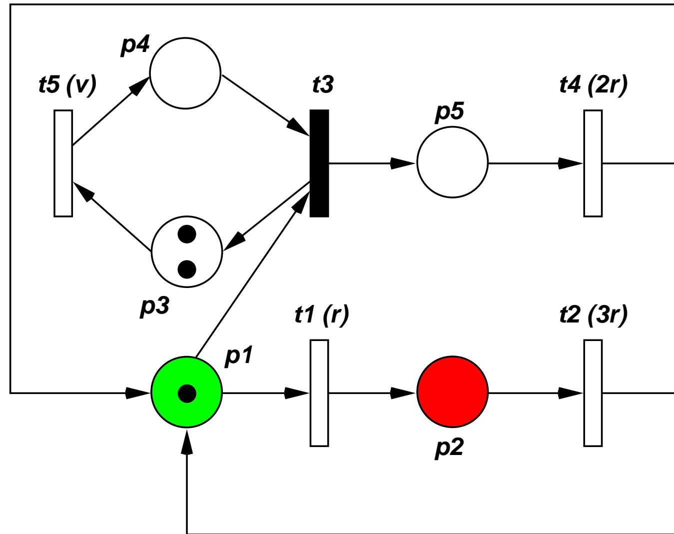

Fig. 3.1. The Simple GSPN model.

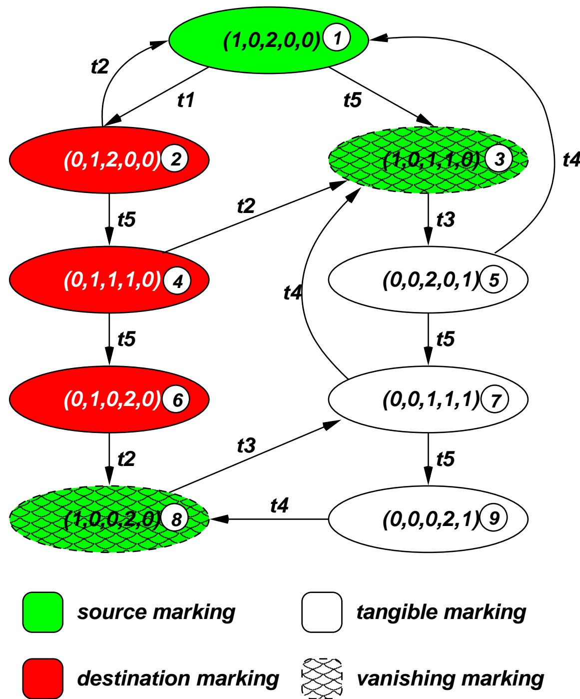

Fig. 3.2. The reachability graph of the Simple GSPN model.

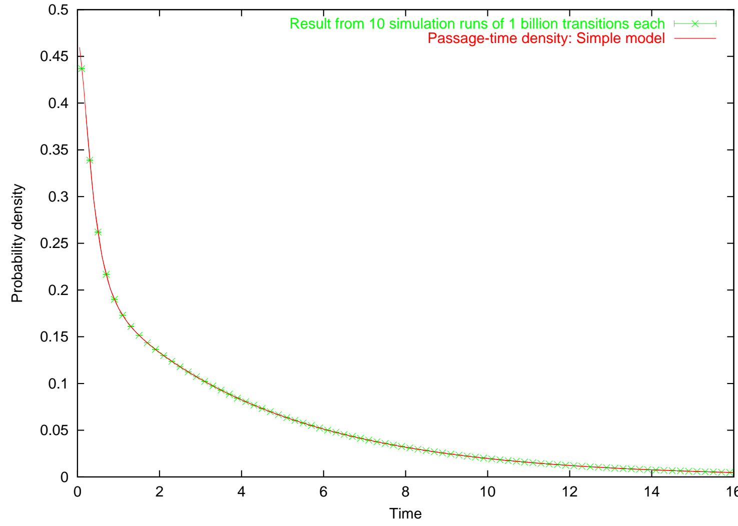

Fig. 3.3. Numerical and simulated response time densities for the Simple model for time taken from markings where $M(p_1) > 0$ to markings where $M(p_2) > 0$. Here the transition rate parameters are $r = 2$ and $v = 5$.

time from these source markings into the destination markings need to be weighted according to the normalised steady state probabilities of the source markings in the GSPN's EMC. Hence, for source markings $M_1 = (1, 0, 2, 0, 0)$, $M_3 = (1, 0, 1, 1, 0)$ and $M_8 = (1, 0, 0, 2, 0)$,

$$L_{\vec{i}\vec{j}}(s) = \alpha_1 L_{1\vec{j}}(s) + \alpha_3 L_{3\vec{j}}(s) + \alpha_8 L_{8\vec{j}}(s)$$

where $\vec{i} = \{1, 3, 8\}$, $\vec{j} = \{2, 4, 6\}$, $\alpha_1 = 0.22952$, $\alpha_3 = 0.43660$ and $\alpha_8 = 0.33388$ (to 5 s.f.).

Fig. 3.3 shows the resulting numerical passage time density. Also shown is the passage time density produced by a discrete-event simulator. This is the combination of the results from 10 runs, each consisting of 1 billion (10^9) transition firings, and the resulting confidence intervals are very narrow indeed. There is excellent agreement between the numerical and simulated densities.

For this small example, a single processor (a PC with a 1.4GHz Athlon processor and 256MB RAM) required just 18 seconds to calculate the 1600 points plotted on the numerical passage time density. The Laguerre method was used and no scaling was needed. This required the solution of 402 sets of linear equations (2 of which were

necessary to determine that no scaling was required, with the remaining 400 used to compute the q_n coefficients).

3.3 Uniformization

As well as the Laplace transform approach described above, passage time densities and quantiles in CTMCs may also be computed through the use of *uniformization* (also known as *randomization*). This transforms a CTMC into one in which all states have the same mean holding time $1/q$, by allowing 'invisible' transitions from a state to itself. This is equivalent to a discrete-time Markov chain, after normalisation of the rows, together with an associated Poisson process of rate q. The one-step transition probability matrix $\mathbf{P}$ which characterises the one-step behaviour of the uniformized DTMC is derived from the generator matrix $\mathbf{Q}$ of the CTMC as:

$$\mathbf{P} = \mathbf{Q}/q + \mathbf{I} \tag{3.7}$$

where the rate $q > \max_i |q_{ii}|$ ensures that the DTMC is aperiodic by guaranteeing that there is at least one single-step transition from a state to itself.

3.3.1 Uniformization for Transient Analysis of CTMCs

Uniformization has classically been used to conduct transient analysis of finite-state CTMCs [66, 112]. The transient state distribution of a CTMC, $\pi_{i\vec{j}}$, is the probability that the process is in a state in $\vec{j}$ at time t, given that it was in state i at time 0:

$$\pi_{i\vec{j}}(t) = \mathbb{P}(\chi(t) \in j \mid \chi(0) = i)$$

where $\chi(t)$ denotes the state of the CTMC at time t.

In a uniformized CTMC, the probability that the process is in state j at time t is calculated by conditioning on $N(t)$, the number of transitions in the DTMC that occur in a given time interval [0,t] [112]:

$$\pi_{i\vec{j}}(t) \;=\; \sum_{m=0}^{\infty} \mathbb{P}\big(\chi(t) \in \vec{j} \mid N(t) = m\big)\mathbb{P}\big(N(t) = m\big)$$

where $N(t)$ is given by a Poisson process with rate q and the state of the uniformized process at time t is denoted $\chi(t)$. Therefore:

$$\pi_{i\vec{j}}(t) \quad = \quad \sum_{n=1}^{\infty} \left(\frac{(qt)^n e^{-qt}}{n!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right)$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P} \qquad \text{for } n \geq 0$$

and $\boldsymbol{\pi}^{(0)}$ is the initial probability distribution from which the transient measure will be measured (typically, for a single initial state i, $\pi_k = 1$ if $k = i$, 0 otherwise).

3.3.2 Uniformization for Passage Time Analysis of CTMCs

Uniformization can also be employed for the calculation of passage time densities in Markov chains as described in [20, 99, 102, 105]. We ensure that only the first passage time density is calculated and that we do not consider the case of successive visits to a target state by making the target states absorbing. We denote by $\mathbf{P}'$ the one-step transition probability matrix of the modified, uniformized chain.

The calculation of the first passage time density between two states has two main components. The first considers the time to complete n hops ($n = 1, 2, 3, \ldots$). Recall that in the uniformized chain all transitions occur with rate q. The density of the time taken to move between two states is found by convolving the state holding-time densities along all possible paths between the states. In a standard CTMC, convolving holding times in this manner is non-trivial as, although they are all exponentially distributed, their rate parameters are different. In a CTMC which has undergone uniformization, however, all states have exponentially-distributed state holding-times with the same parameter q. This means that the convolution of n of these holding-time densities is the convolution of n exponentials all with rate q, which is an n-stage Erlang density with rate parameter q.

Secondly, it is necessary to calculate the probability that the transition between a source and target state occurs in exactly n hops of the uniformized chain, for every value of n between 1 and a maximum value m. The value of m is determined when

the value of the nth Erlang density function (the left-hand term in Eq. 3.8) drops below some threshold value. After this point, further terms are deemed to add nothing significant to the passage time density and so are disregarded.

The density of the time to pass between a source state i and a target state j in a uniformized Markov chain can therefore be expressed as the sum of m n-stage Erlang densities, weighted with the probability that the chain moves from state i to state j in exactly n hops ($1 \leq n \leq m$). This can be generalised to allow for multiple target states in a straightforward manner; when there are multiple source states it is necessary to provide a probability distribution across this set of states (such as the renormalised steady-state distribution calculated below in Eq. 3.10).

The response time between the non-empty set of source states $\vec{i}$ and the non-empty set of target states $\vec{j}$ in the uniformized chain therefore has probability density function:

$$\begin{aligned} f_{\vec{i}\vec{j}}(t) &= \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \\ &\simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \end{aligned} \tag{3.8}$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P}' \qquad \text{for } n \geq 0 \tag{3.9}$$

with

$$\pi_k^{(0)} = \begin{cases} 0 & \text{for } k \notin \vec{i} \\ \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{for } k \in \vec{i} \end{cases} \tag{3.10}$$

The π_k values are the steady state probabilities of the corresponding state k from the CTMC's embedded Markov chain. When the convergence criterion

$$\frac{||\boldsymbol{\pi}^{(n)} - \boldsymbol{\pi}^{(n-1)}||_\infty}{||\boldsymbol{\pi}^{(n)}||_\infty} < \varepsilon \tag{3.11}$$

is met, for given tolerance ε, the vector $\boldsymbol{\pi}^{(n)}$ is considered to have converged and no further multiplications with $\mathbf{P}'$ are performed. Here, $||\mathbf{x}||_\infty$ is the infinity-norm given by $||\mathbf{x}||_\infty = \max_i |x_i|$.

The corresponding cumulative distribution function for the passage time, $F_{\vec{i}\vec{j}}(t)$, can

be calculated by substituting the cumulative distribution function for the Erlang distribution into Eq. 3.8 in place of the Erlang density function term, viz.:

$$\begin{aligned} F_{\vec{i}\vec{j}}(t) &= \sum_{n=1}^{\infty} \left(\left(1 - e^{-qt} \sum_{k=0}^{n-1} \frac{(qt)^k}{k!} \right) \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \\ &\simeq \sum_{n=1}^{m} \left(\left(1 - e^{-qt} \sum_{k=0}^{n-1} \frac{(qt)^k}{k!} \right) \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \end{aligned}$$

where $\boldsymbol{\pi}^{(n)}$ is defined as in Eqs. 3.9 and 3.10.

3.4 Comparison of Methods

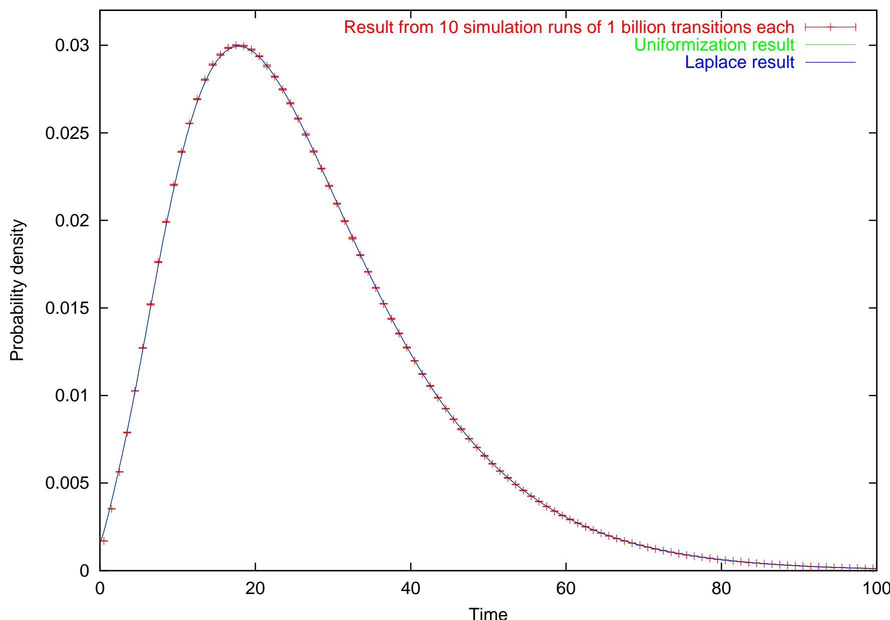

Fig. 3.4. Numerical and simulated (with 95% confidence intervals) passage time densities for time taken from the initiation of a transport layer transmission to the arrival of an acknowledgement packet in the Courier model.

As both the Laplace transform method and uniformization can be used to calculate passage time densities in Markov models, it is instructive to compare the run-time performance of the two methods along with simulation. Table 3.1 shows the run-times in seconds taken to compute the passage time densities shown in Figs. 3.4, 3.5

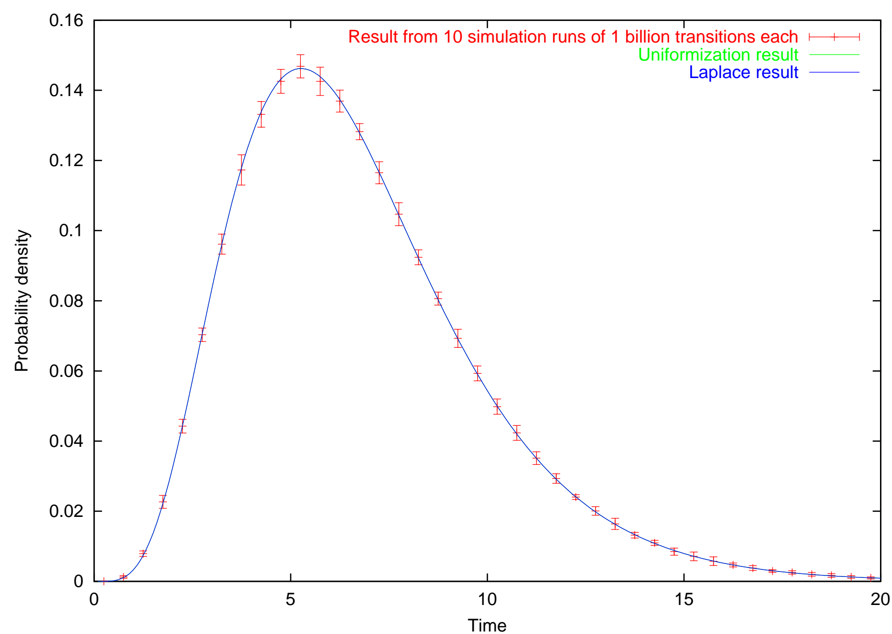

Fig. 3.5. Numerical and simulated (with 95% confidence intervals) density for the time taken to produce a finished part of type $P12$ starting from states in which there are $k = 6$ unprocessed parts of types $P1$ and $P2$ in the FMS model.

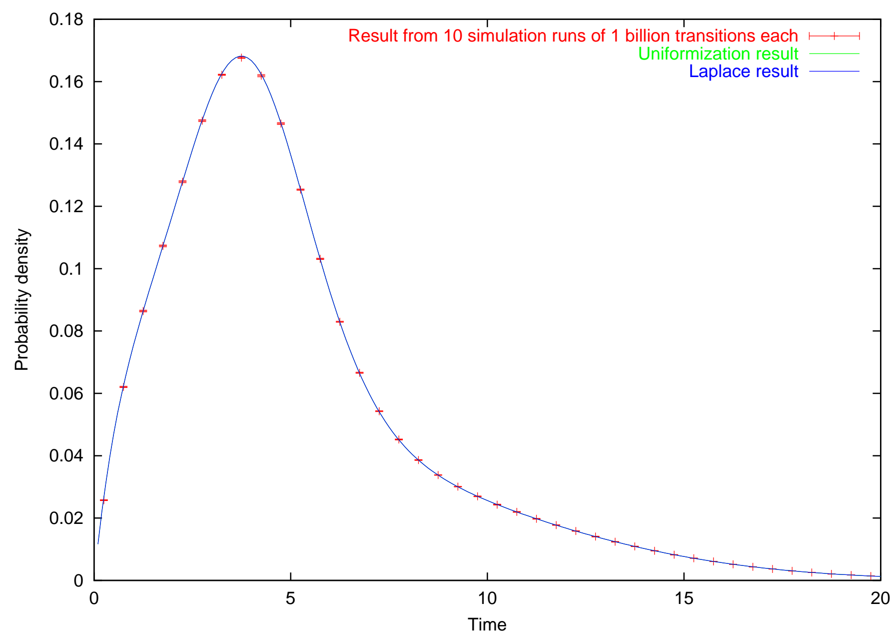

Fig. 3.6. Numerical and simulated (with 95% confidence intervals) passage time densities for the cycle-time in a tree-like queueing network with 15 customers.

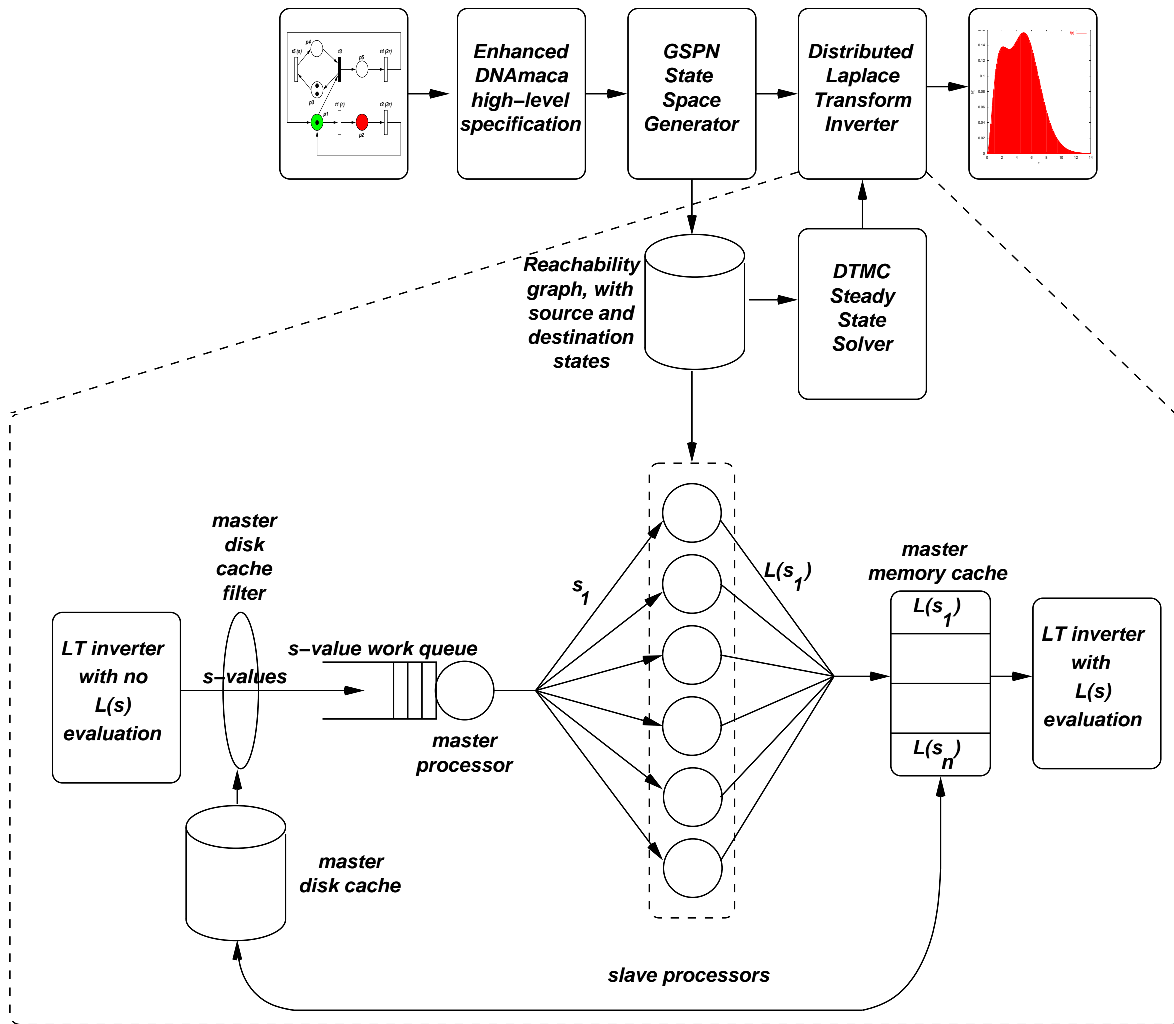

Fig. 3.7. GSPN passage time density calculation pipeline [48].

Model Name	No. of States	Uniform. Run-time	Laplace Run-times						Sim. Run-time (1 run)
			1 PC	2 PCs	4 PCs	8 PCs	16 PCs	32 PCs	
Courier	11 700	1.9	42.1	30.0	22.4	19.6	18.0	23.1	3 656.0
FMS	537 768	64.8	5 096.0	2 582.6	1 298.4	675.8	398.4	182.1	2 729.6
Tree	542 638	126.2	7 555.3	4 719.3	1 921.9	993.0	550.9	398.6	1 976.8

Table 3.1. Comparison of run-time in seconds for uniformization, Laplace transform inversion and simulation passage time analysis.

Model Name	No. of States	Laplace Run-times					
		1 PC	2 PCs	4 PCs	8 PCs	16 PCs	32 PCs
Courier	29 010	542.7	293.6	170.6	145.6	166.6	232.8
FMS	2 519 580	27 593.8	13 790.2	6 961.3	3 548.9	1 933.4	1 079.7

Table 3.2. Run-time in seconds for the Laplace transform inversion method on GSPN state-spaces without vanishing state elimination.

and 3.6 using uniformization, Laplace transform inversion and simulation. The run-times were produced on a network of PC workstations linked together by 100Mbps switched Ethernet, each PC having an Intel Pentium 4 2.0GHz processor and 512MB RAM.

3.4.1 Models Studied

Results are presented for three models: a GSPN model of a communication protocol, a GSPN model of a flexible manufacturing system and a tree-like queueing network. These models are described in detail in Appendices A.1, A.2 and A.3 respectively. In order for uniformization to be used and compared fairly with the Laplace transform method, it was necessary in the case of the two GSPN models to generate the state spaces with the vanishing states eliminated – the number of states given in the second column of Table 3.1 is therefore the size of each model's underlying CTMC.

For the Courier protocol model, the passage of interest is from markings for which $M(p_{11}) > 0$ to those markings for which $M(p_{20}) > 0$. This corresponds to the end-

to-end response time from the initiation of a transport layer transmission to the arrival of the corresponding acknowledgement packet; as the sliding window size n is set to 1, there can be only one outstanding unacknowledged packet at any time. In the Flexible Manufacturing System (FMS) model we calculate the density of the time taken to produce a finished part of type $P12$ starting from any state in which there are 6 unprocessed parts of type $P1$ and 6 unprocessed parts of type $P2$. That is, the source markings are those where $M(P1) = M(P2) = 6$ and the target markings are those where $M(P12s) = 1$. Finally, for the tree-like queueing network, results are presented for a model with 15 customers and show the density of the cycle time of a customer from when it arrives at the back of the queue for server $q1$ to when it reaches that point again.

For uniformization and simulation, the results were produced using a single PC, but for the Laplace transform method a parallel solver illustrated in Fig. 3.7 was used. This has a distributed master–slave architecture, where the master processor calculates in advance at which values of s Eq. 3.3 or Eq. 3.6 will need to be solved in order to perform the numerical inversion. These values of s are placed in a queue to which slave processors make requests. They are allocated the next s value available and then construct and solve the set of linear equations for that value of s, returning the result to the master to be cached. When all the results have been returned, the master processor then uses the cached values to perform the inversion and returns the value of the passage time density or distribution at the required values of t.

The simulation results presented in the graphs are the combined results from 10 runs, each run consisting of 1 billion (10^9) transition firings. The timing information for simulation in Table 3.1 is the average time taken to perform one of these runs: as the runs are independent of each other they can be executed in parallel and so 10 runs on 10 machines should take no longer than 1 run on 1 machine.

3.4.2 Discussion

From Table 3.1 it can be seen that uniformization (running on a single processor) is much faster than the Laplace transform method (for all number of processors up to 32)

except in the case of the smallest model considered (Courier with $11\,700$ states). Using the Laguerre method required the solution of 402 sets of equations of the form of Eq. 3.4 for the tree-like queueing network and FMS models, and 804 sets for Courier. The reason that Courier required more equations to be solved was the use of the scaling technique referred to in Section 2.3.2. We also note that for the Courier model the solution took longer on 32 machines than it did on 16. This can be attributed to increased contention for the global work queue.

In contrast, the uniformization implementation needed only to perform a single set of sparse matrix–vector multiplications of the form shown in Eq. 3.9. It must be noted, however, that the Laplace transform method is easier to extend to systems with generally-distributed state holding-time distributions (see Chapter 4 below) and preserves the ability to reason about source and target states which are vanishing. This ability is lost in the uniformization method as vanishing states must be eliminated when generating the state-space for the method to function.

Table 3.1 also shows that a single simulation run took much longer than either uniformization or the Laplace transform method for all three models and 10 runs only produced inexact results bounded by confidence intervals (although the intervals are fairly tight in two of the three cases). This run-time could, however, have been reduced by performing fewer transition firings but this may have resulted in wider confidence intervals. Simulation may not be suitable for passage time calculation in all models, particularly those which are very large but have very few initial states. In such cases, many more transition firings may have to be performed in order to achieve meaningful results as the number of observed passages would otherwise be too low. By way of example, the CTMC underlying the FMS model has $537\,638$ states, of which only 28 are source states (0.005% of the total states) and $136\,584$ are target states. The CTMC of the Courier model with $11\,700$ states has $3\,150$ source states (26.9% of the total states) and 900 target states. We observe that the confidence intervals on the Courier passage time density of Fig. 3.4 are much tighter than those on the FMS density in Fig. 3.5.

When the method of Section 3.2 is used for the analysis of GSPN models, the underlying state-spaces are larger as vanishing states are not eliminated. Table 3.2 shows

the sizes of the state-spaces for the two GSPN models and the time taken to analyse them for the same passage time quantities as Table 3.1 using the Laplace transform method for GSPNs. Note the large increase in the size of the underlying process when vanishing states are not eliminated (it has more than doubled in the case of Courier and increased by a factor of 5 for the FMS model) and the consequent increase in time taken to compute the results. This illustrates that, although the Laplace transform method for GSPNs offers the opportunity to reason about vanishing source and target states, the modeller must be aware that it does so at increased computational cost.

We note that we have developed a parallel tool called HYDRA (described in Chapter 6) which implements the uniformization method. This was not used here, however, as the size of the state spaces under consideration made it unnecessary. Its use could reduce the time taken to perform the uniformization calculations even further – the results presented in Section 8.1.3 demonstrate how well such an implementation scales.

3.5 Passage Times in Stochastic Process Algebras

The main focus of the work in this chapter is on the calculation of passage times in CTMCs derived from SPNs or GSPNs. The techniques described can, however, be applied to CTMCs generated from other high-level formalisms – in Section 3.4 above analysis is conducted on a queueing network. Stochastic process algebras are another popular modelling formalism, and the tools and techniques presented in this thesis can easily be applied to their analysis for passage time and transient measures.

We focus here on models specified in PEPA (described in Section 2.2.5). Using the Imperial PEPA Compiler (ipc) [22, 23], PEPA models can be converted into equivalent Petri net models. It also permits the expression of passage time queries in terms of the actions in the original PEPA model through the use of *stochastic probes*. These are fragments of process algebra which observe the behaviour of the model and change state to indicate that actions marking the start or end of the passage time measure of interest have occurred [6]. Interested readers are directed to [22, 23] for full details of ipc's implementation.

$$
\begin{aligned}
Person_1 &\stackrel{\text{def}}{=} (reg_1, r).Person_1 + (move_2, m).Person_2 \\
Person_i &\stackrel{\text{def}}{=} (move_{i-1}, m).Person_{i-1} + (reg_i, r).Person_i \\
&\quad +(move_{i+1}, m).Person_{i+1} \quad : 1 < i < N \\
Person_N &\stackrel{\text{def}}{=} (move_{N-1}, m).Person_{i-1} + (reg_N, r).Person_N \\
\\
Sensor_i &\stackrel{\text{def}}{=} (reg_i, \top).(rep_j, s).Sensor_i \quad : 1 \leq i \leq N \\
\\
Dbase_i &\stackrel{\text{def}}{=} \sum_{j=1}^{N} (rep_j, \top).Dbase_j \quad : 1 \leq i \leq N \\
\\
Sys &\stackrel{\text{def}}{=} \prod_{j=1}^{M} Person_1 \underset{Reg}{\bowtie} \prod_{j=1}^{N} Sensor_j \underset{Rep}{\bowtie} Dbase_1 \\
\\
&\text{where } Reg = \{reg_i \mid 1 \leq i \leq N\} \\
&\quad \text{and } Rep = \{rep_i \mid 1 \leq i \leq N\}
\end{aligned}
$$

Fig. 3.8. The PEPA description for the generalised active badge model with N rooms and M people [23].

3.5.1 Example of SPA Analysis

We illustrate the analysis of PEPA models for passage time quantities with a small example of an active badge system. In the original model described in [65], there are 4 rooms on a corridor all installed with active badge sensors and a single person who can move from one room to an adjacent room. The sensors are linked to a database which records which sensor has been activated last.

In the model of Fig. 3.8 (reproduced from [23]), we have extended this to support M people in N rooms with sensors and a database that can be in one of N states, representing the last sensor to be activated.

In the model, $Person_i$ represents a person in room i, $Sensor_i$ is the sensor in room

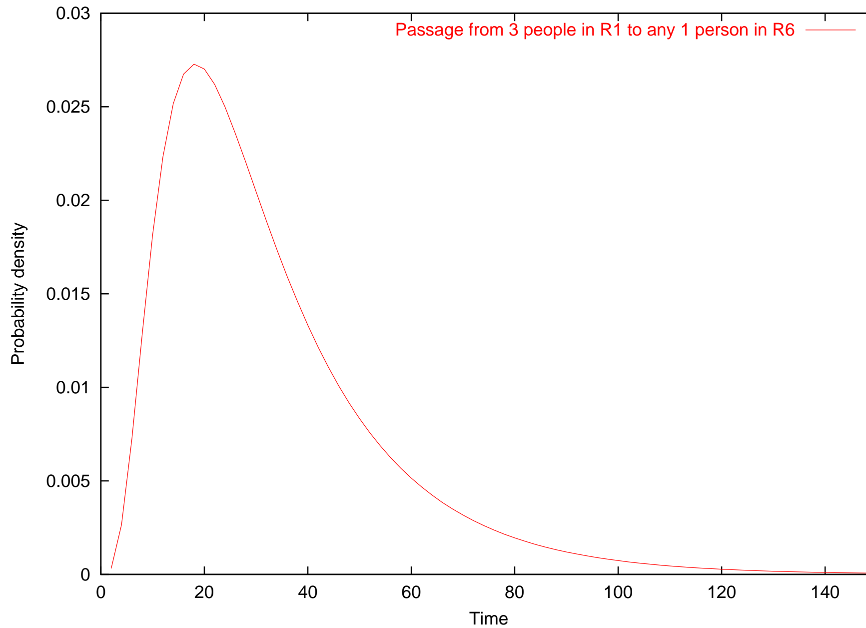

Fig. 3.9. Passage time density of the time taken for the first person to move from room 1 to room 6 in the 3-person Active Badge model.

i and $Dbase_i$ is the state of the database. A person in room i can either move to room $i-1$ or $i+1$ or, if they remain there long enough, set off the sensor in room i, which registers its activation with the database. Allowing M people in N rooms yields N^M different configurations. Independently, there are 2^N sensor configurations and N states that the database can be in, giving a total of $2^N N^{M+1}$ states.

For 3 people and 6 rooms, we have a global state space of $82\,944$ states. Fig. 3.9 shows the passage time density of the time taken for the first person to move from room 1 to room 6. This was calculated using uniformization.

3.6 Estimation of Passage Time Densities and Distributions From Their Moments

As well as calculating full passage time densities in Markov models we can also compute the moments of such densities. Not only are these meaningful performance measures in their own right, but it is possible to estimate the full density or distribution

from a small number of its moments at less computational cost than calculating the full distribution using the techniques described above.

The nth raw moment $M_{i\vec{j}}(n)$ of the passage time quantity $P_{i\vec{j}}$ is obtained by differentiating the corresponding Laplace transform $L_{i\vec{j}}(s)$ n times and evaluating the resulting expression at $s = 0$. For example, for $n = 1$:

$$\begin{aligned} f(s) &= \int_0^\infty e^{-st} f(t)\, dt \\ \implies \quad f'(s) &= -\int_0^\infty te^{-st} f(t)\, dt \\ \implies \quad f'(0) &= -\int_0^\infty tf(t)\, dt \\ \implies \quad M_{i\vec{j}}(1) &= -f'(0) \end{aligned} \tag{3.12}$$

In general:

$$f'^{(n)}(0) = (-1)^n \int_0^\infty t^n f(t)\, dt$$

In terms of $M_{i\vec{j}}(n)$ and $L_{i\vec{j}}(s)$ [70]:

$$M_{i\vec{j}}(n) = (-1)^n \left. \frac{d^n L_{i\vec{j}}(s)}{ds^n} \right|_{s=0} \tag{3.13}$$

3.6.1 Moment Calculation for CTMCs

The general formula for calculating the nth moment of passage time for a Markov model is stated without proof in [70]:

$$-q_{ii} M_{i\vec{j}}(n) = \sum_{k \notin \vec{j}} q_{ik} M_{k\vec{j}}(n) + n M_{i\vec{j}}(n-1) \tag{3.14}$$

for $i \notin \vec{j}$, $M_{i\vec{j}}(n) = 0$ for $i \in \vec{j}$ and $M_{i\vec{j}}(0) = 1$. We here prove that this holds by induction.

We have shown in Section 3.1 that to calculate the Laplace transform of the density of the passage time between states i and states $\vec{j}$ we solve a system of linear equations of the form:

$$(s - q_{ii}) L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} q_{ik} L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} q_{ik} \tag{3.15}$$

where q_{ik} is the instantaneous rate between state i and k, and q_{ii} is the negated sum of all the rates out of state i. We will now show that differentiating Eq. 3.15 n times with respect to s gives:

$$(s - q_{ii})L_{i\vec{j}}^{(n)'}(s) + nL_{i\vec{j}}^{(n-1)'}(s) = \sum_{k \notin \vec{j}} q_{ik} L_{k\vec{j}}^{(n)'}(s) \tag{3.16}$$

for any integer value of n. Recall that the product rule for differentiating two functions $f(x)$ and $g(x)$ states:

$$(fg)'(x) = f(x)g'(x) + f'(x)g(x)$$

Base case Differentiating Eq. 3.15 once requires the use of the product rule, and this yields:

$$(s - q_{ii})L'_{i\vec{j}}(s) + L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} q_{ik} L'_{k\vec{j}}(s)$$

Which is Eq. 3.16 for $n = 1$.

Inductive step Given that Eq. 3.16 holds, we must show that differentiating it once gives Eq. 3.16 for the $(n + 1)$ differential. Applying the product rule we have:

$$\begin{aligned} (s - q_{ii})L_{i\vec{j}}^{(n+1)'}(s) + L_{i\vec{j}}^{(n)'}(s) + nL_{i\vec{j}}^{(n)'}(s) &= \sum_{k \notin \vec{j}} q_{ik} L_{k\vec{j}}^{(n+1)'}(s) \\ \implies (s - q_{ii})L_{i\vec{j}}^{(n+1)'}(s) + (n+1)L_{i\vec{j}}^{(n)'}(s) &= \sum_{k \notin \vec{j}} q_{ik} L_{k\vec{j}}^{(n+1)'}(s) \end{aligned}$$

as required.

We have therefore proved that Eq. 3.16 holds for all integer values of $n \geq 1$. Evaluating this at $s = 0$ then yields the expression for the nth moment of the passage time, Eq. 3.14, as required.

In terms of computational requirements, calculating n moments of a passage time density requires the solution of n sets of $N \times N$ linear equations, one for each moment, each of the form of Eq. 3.14.

3.6.2 Extension to GSPNs

The formulae above for the calculation of moments in CTMCs can be extended to the calculation of moments in GSPNs in an analogous manner to the way in which the calculation of passage time densities in CTMCs was extended to passage times in GSPNs. Once again, it is necessary to take into consideration the possible existence of vanishing markings. Recall that the Laplace transform of the passage time density from a vanishing marking i to a set of target states $\vec{j}$ is given by:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} p_{ik} L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} \tag{3.17}$$

By differentiating Eq. 3.17 with respect to s once we get:

$$L'_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} p_{ik} L'_{k\vec{j}}(s)$$

Evaluating this at $s = 0$ to calculate the first moment gives:

$$M_{i\vec{j}}(1) = \sum_{k \notin \vec{j}} p_{ik} M_{k\vec{j}}(1)$$

Higher moments can be calculated by repeated differentiation of Eq. 3.17. In general, to calculate the nth moment of a passage time from a vanishing state i to a set of target states $\vec{j}$:

$$M_{i\vec{j}}(n) = \sum_{k \notin \vec{j}} p_{ik} M_{k\vec{j}}(n) \tag{3.18}$$

The first n moments of a passage time in a GSPN model starting from a vanishing state i can therefore be calculated by using Eq. 3.14 and Eq. 3.18:

$$M_{i\vec{j}}(n) = \begin{cases} \sum_{k \notin \vec{j}} \frac{q_{ik}}{-q_{ii}} M_{k\vec{j}}(n) + \frac{1}{-q_{ii}} n M_{i\vec{j}}(n-1) & \text{if } i \in \mathcal{T} \\ \\ \sum_{k \notin \vec{j}} p_{ik} M_{k\vec{j}}(n) & \text{if } i \in \mathcal{V} \end{cases}$$

3.6.3 Distribution Estimation from Moments

The reconstruction of a function based on its first n moments is a well-known problem. It is of particular interest in passage time analysis as the calculation of the moments

of a passage time density can be accomplished at much less computational cost than the calculation of the full density – in a CTMC, the first four moments are obtained by solving four sets of linear equations of the type shown in Eq. 3.14, while the full density would require the solution of perhaps 400 sets of linear equations of the type shown in Eq. 3.3 (if using the Laguerre inversion method). Once the first few moments have been calculated, there are a number of techniques by which functions can be approximated. We describe here briefly a technique based on the use of the Generalised Lambda Distribution (GLD) [7, 95].

The Generalised Lambda Distribution

The Generalised Lambda Distribution (GLD) is a curve capable of assuming a wide variety of different shapes depending on the value of its parameters. For passage time analysis, the challenge lies in determining the parameter values which result in a GLD curve that approximates well a given passage time density or distribution. The GLD is defined by a quantile (inverse cumulative distribution) function $Q(u)$ which has four parameters λ_1, λ_2, λ_3 and λ_4. The first is the location parameter, the second the scale parameter and the third and fourth are the shape parameters. $Q(u)$ is defined in terms of these four parameters as [59]:

$$Q(u) = \lambda_1 + \frac{1}{\lambda_2}\left(\frac{u^{\lambda_3}-1}{\lambda_3} - \frac{(1-u)^{\lambda_4}-1}{\lambda_4}\right) \tag{3.19}$$

The expression of $Q(u)$ in this form is known as the FKML parameterisation [59]. The process of approximating a passage time density using the GLD involves computing values for these four parameters such that the first four moments of the resulting GLD (mean μ, variance σ^2, skewness α_3 and kurtosis α_4) match the first four moments of the passage time measure ($\hat{\mu}$, $\hat{\sigma}^2$, $\hat{\alpha_3}$ and $\hat{\alpha_4}$ respectively).

As $Q(u)$ is a quantile function it will yield the value of x such that $F(x)$ (the corresponding cumulative distribution function) equals u. The probability density function $f(x)$ of the GLD can therefore be calculated as:

$$f(x) = \frac{du}{dx} = \frac{du}{dQ(u)} = \left(\frac{dQ(u)}{du}\right)^{-1}$$

It follows that the kth raw moment of a random variable χ with quantile function $Q(u)$ can be calculated as:

$$\begin{aligned} E[\chi^k] &= \int_0^\infty x^k f(x)\, dx \\ &= \int_0^1 (Q(u))^k \frac{du}{dQ(u)}\, dQ(u) \\ &= \int_0^1 (Q(u))^k\, du \end{aligned} \tag{3.20}$$

Expanding Eq. 3.19 gives [95]:

$$\begin{aligned} Q(u) &= \left(\lambda_1 - \frac{1}{\lambda_2\lambda_3} + \frac{1}{\lambda_2\lambda_4} + \frac{1}{\lambda_2}\left(\frac{u^{\lambda_3}}{\lambda_3} - \frac{(1-u)^{\lambda_4}}{\lambda_4}\right)\right) \\ &= a + bR(u) \end{aligned}$$

where

$$R(u) = \left(\frac{u^{\lambda_3}}{\lambda_3} - \frac{(1-u)^{\lambda_4}}{\lambda_4}\right)$$

The first four central moments $\hat{q}_k$, $1 \le k \le 4$, of $Q(u)$ can then be expressed in terms of the first four raw moments r_k, $1 \le k \le 4$, of $R(u)$:

$$\begin{aligned} \hat{q}_1 &= \lambda_1 - \frac{1}{\lambda_2\lambda_3} + \frac{1}{\lambda_2\lambda_4} + \frac{r_1}{\lambda_2} \\ \hat{q}_2 &= \frac{1}{\lambda_2^2}(r_2 - r_1^2) \\ \hat{q}_3 &= \frac{1}{\lambda_2^3}(r_3 - 3r_1r_2 + 2r_1^3) \\ \hat{q}_4 &= \frac{1}{\lambda_2^4}(r_4 - 4r_1r_3 + 6r_1^2r_2 - 3r_1^4) \end{aligned} \tag{3.21}$$

The kth raw moment of $R(u)$ can be calculated in exactly the same manner as for $Q(u)$ in Eq. 3.20:

$$r_k = \int_0^1 \left(\frac{u^{\lambda_3}}{\lambda_3} - \frac{(1-u)^{\lambda_4}}{\lambda_4}\right)^k du$$

Using binomial expansion (as in [95]) yields:

$$\begin{aligned} r_k &= \int_0^1 \sum_{j=0}^{k} \binom{k}{j} (-1)^j \frac{u^{\lambda_3(k-j)}}{\lambda_3^{k-j}} \frac{(1-u)^{\lambda_4 j}}{\lambda_4^j}\, du \\ &= \sum_{j=0}^{k} \frac{(-1)^j}{\lambda_3^{k-j}\lambda_4^j} \binom{k}{j} \beta(\lambda_3(k-j)+1, \lambda_4 j + 1) \end{aligned} \tag{3.22}$$

where

$$\beta(a,b) = \int_0^1 u^{a-1}(1-u)^{b-1}\,du$$

As r_k is defined only for positive arguments, it is required that $\min(\lambda_3, \lambda_4) > -1/k$. Expanding Eq. 3.22 for $k = 1, 2, 3, 4$ gives:

$$\begin{aligned}
r_1 &= \frac{1}{\lambda_3(\lambda_3+1)} - \frac{1}{\lambda_4(\lambda_4+1)} \\
r_2 &= \frac{1}{\lambda_3^2(2\lambda_3+1)} + \frac{1}{\lambda_4^2(2\lambda_4+1)} - \frac{2}{\lambda_3\lambda_4}\beta(\lambda_3+1, \lambda_4+1) \\
r_3 &= \frac{1}{\lambda_3^3(3\lambda_3+1)} - \frac{1}{\lambda_4^3(3\lambda_4+1)} - \frac{3}{\lambda_3^2\lambda_4}\beta(2\lambda_3+1, \lambda_4+1) \\
&\quad + \frac{3}{\lambda_3\lambda_4^2}\beta(\lambda_3+1, 2\lambda_4+1) \\
r_4 &= \frac{1}{\lambda_3^4(4\lambda_3+1)} + \frac{1}{\lambda_4^4(4\lambda_4+1)} + \frac{6}{\lambda_3^2\lambda_4^2}\beta(2\lambda_3+1, 2\lambda_4+1) \\
&\quad - \frac{4}{\lambda_3^3\lambda_4}\beta(3\lambda_3+1, \lambda_4+1) - \frac{4}{\lambda_3\lambda_4^3}\beta(\lambda_3+1, 3\lambda_4+1)
\end{aligned} \tag{3.23}$$

The skewness α_3 of $Q(u)$ is therefore given by Eq. 3.21 and Eq. 3.23:

$$\begin{aligned}
\alpha_3 &= \frac{1}{\sigma^3}E[\chi - E[\chi]]^3 \\
&= \frac{\hat{q}_3}{\hat{q}_2^{\,3/2}} \\
&= \frac{r_3 - 3r_1r_2 + 2r_1^3}{(r_2 - r_1^2)^{3/2}}
\end{aligned}$$

Similarly, the kurtosis α_4 of $Q(u)$ is calculated from:

$$\begin{aligned}
\alpha_4 &= \frac{1}{\sigma^4}E[\chi - E[\chi]]^4 \\
&= \frac{\hat{q}_4}{\hat{q}_2^{\,2}} \\
&= \frac{r_4 - 4r_1r_3 + 6r_1^2r_2 - 3r_1^4}{(r_2 - r_1^2)^2}
\end{aligned}$$

As we require the skewness and kurtosis of the GLD to match those of the passage time quantity, we can obtain values for λ_3 and λ_4 by setting $\alpha_3 = \hat{\alpha}_3$ and $\alpha_4 = \hat{\alpha}_4$ and solving the resulting non-linear equations.

Finally, the values of λ_1 and λ_2 are computed using the following equations:

$$\begin{aligned}
\lambda_1 &= \frac{\sqrt{r_2 - r_1^2}}{\hat{\sigma}} \\
\lambda_2 &= \hat{\mu} + \frac{1}{\lambda_2}\left(\frac{1}{\lambda_3+1} - \frac{1}{\lambda_4+1}\right)
\end{aligned}$$

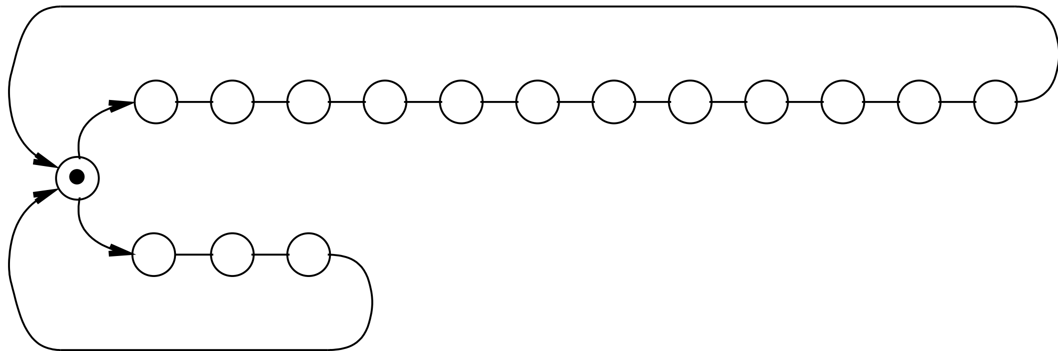

Fig. 3.10. The branching Erlang model.

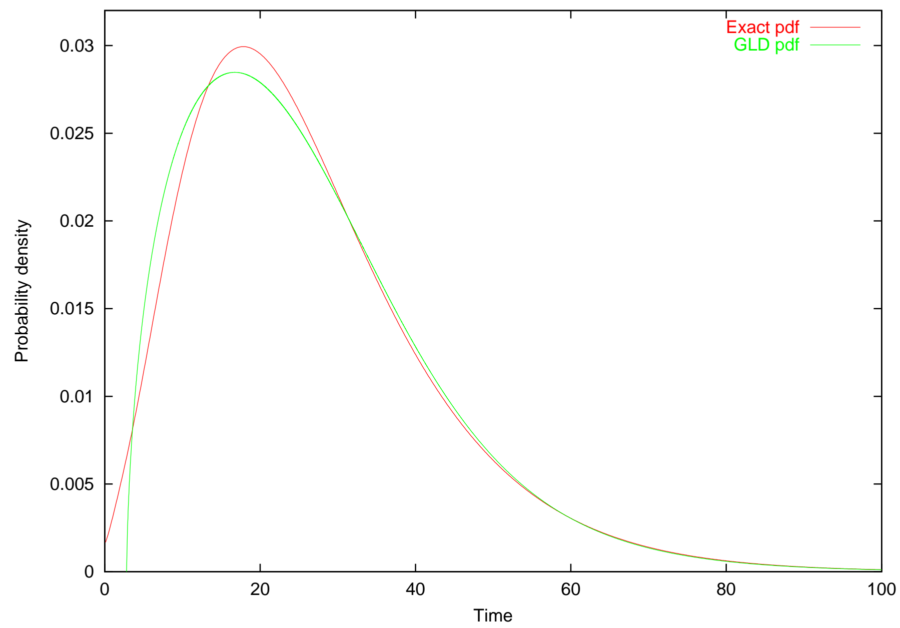

Fig. 3.11. The approximate Courier model passage time density function produced by the GLD method compared with the exact result.

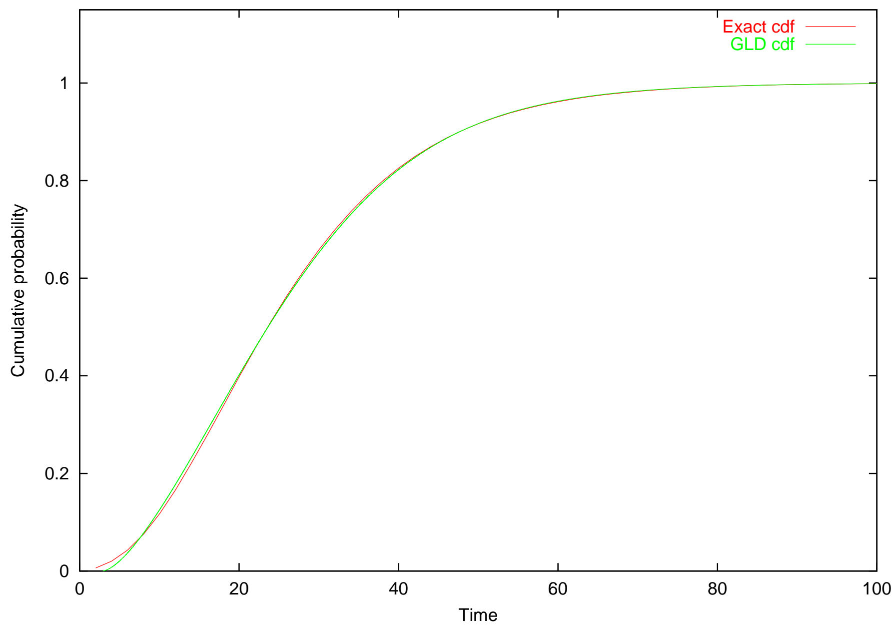

Fig. 3.12. The approximate Courier model cumulative distribution function produced by the GLD method compared with the exact result.

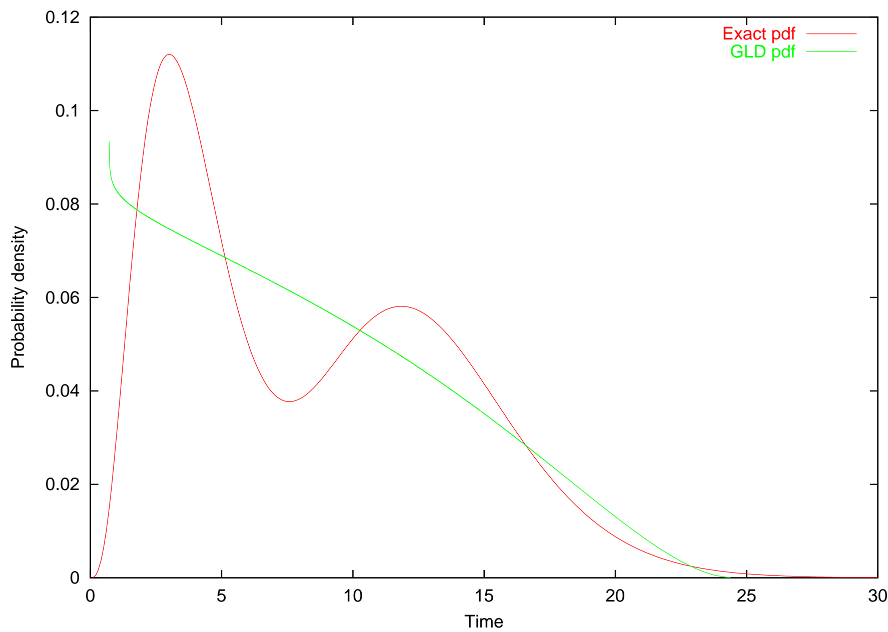

Fig. 3.13. The approximate Erlang model passage time density function produced by the GLD method compared with the exact result.

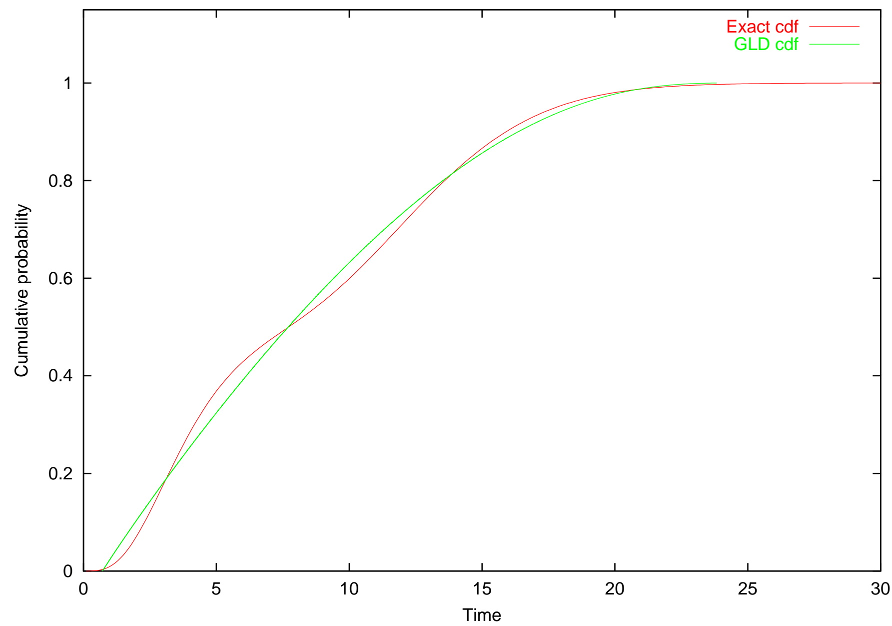

Fig. 3.14. The approximate Erlang model cumulative distribution function produced by the GLD method compared with the exact result.

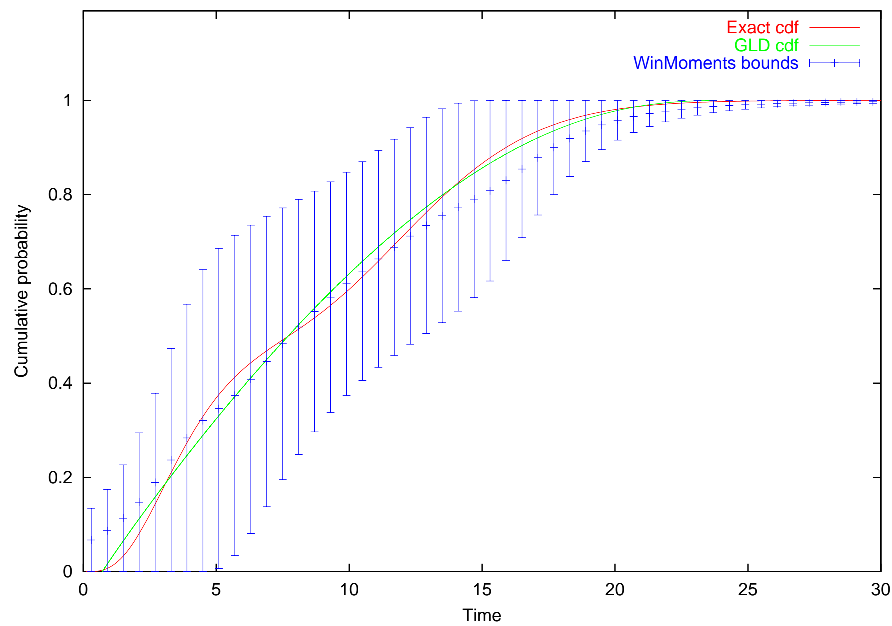

Fig. 3.15. The branching Erlang model cumulative distribution function produced by the GLD method compared with the bounds produced by the WinMoments tool.

An example passage time density function and the approximation produced by the GLD method can be seen in Fig. 3.11 (this is in fact the passage time measure for the Courier model from Section 3.4). This was produced using our implementation of the above GLD method as described in [8]. We notice good agreement between the exact result and the approximation. The corresponding cumulative distribution function (in the case of the GLD method produced by numerical integration of the probability density function but computed directly for the exact passage times) can also be seen in Fig. 3.12. Again, the agreement is good.

One limitation of the GLD method is that it assumes that the function being approximated is unimodal. Attempting to approximate a bimodal passage time density results in a very poor fit. The branching Erlang model shown in Fig. 3.10 is composed of two equiprobable branches, one with an $erlang(1.0, 12)$ distribution and the other an $erlang(1.0, 3)$ distribution. As can be seen in Fig. 3.13, the approximation produced by the GLD method does not capture the bimodal nature of the passage time density. The corresponding cumulative distribution function (shown in Fig. 3.14) does show better agreement, however.

Model	States	Moment Calc.	Moment Matching	GLD Total	Laplace Total
Courier	11 700	0.66	0.28	0.94	42.1
Erlang	32	0.05	0.27	0.32	14.6

Table 3.3. Comparison of run-times in seconds for GLD approximation and full Laplace transform-based passage time solution.

Table 3.3 shows that the time taken to calculate the above passage time densities using the GLD method is considerably less than that required by the Laplace transform-based approach. With increasing model size, we anticipate an increase in the time taken to calculate the moments (although this will still require two orders of magnitude less computation than the Laplace transform approach). The time to approximate the density from these moments should remain very rapid since it is independent of the model size.

A different approach based on the calculation of Hankel determinants is taken in [111], where a technique is presented to calculate the upper and lower bounds on the range of feasible distribution functions given a number of the moments of the function. Fig. 3.15 compares the output produced by our implementation of the GLD method [7, 8] with that of an implementation of this second method (an implementation of the WinMoments tool [111]) for the branching Erlang model. In both cases the approximations are based on the first four moments of the passage time density. It will be observed that the bounds calculated by WinMoments are very wide except for extreme values. Also, in this case the GLD method provides a better approximation to the exact distribution than the mid-points of the WinMoments bounds.

Chapter 4

Passage Times in Semi-Markov Models

In this chapter we present a central contribution of this thesis: an iterative passage time density extraction algorithm for very large semi-Markov processes (SMPs). Previous attempts to analyse SMPs for passage time measures have not been applicable to very large models due to the complexity of maintaining the Laplace transforms of state holding time distributions in closed form. In previous work [64, 98], the time complexity of the numerical calculation of passage time densities and quantiles for a semi-Markov system with N states is $O(N^4)$. Consequently, it has not been possible to analyse semi-Markov systems with more than a few thousand states.

This limitation is overcome here by the application of an efficient representation for the Laplace transforms of the state holding time density functions, which was developed with the demands of the numerical inversion algorithms described in Chapter 2 in mind. The resulting technique is amenable to a parallel implementation (thus allowing for the analysis of even larger semi-Markov models) and has a time complexity of $O(N^2 r)$ for r iterations (typically $r << N$).

We also present an iterative algorithm for computing transient state probabilities in SMPs which requires less computational effort than existing techniques. The algorithm builds on the iterative passage time algorithm and shows a similar time com-

plexity. We present example passage time and transient results from models with up to 1.1 million states. The chapter concludes by considering the extraction of moments of passage times in semi-Markov systems.

4.1 Efficient Representation of General Distributions

The key to practical analysis of semi-Markov processes lies in the efficient representation of their general distributions. Without care the structural complexity of the SMP can be recreated within the representation of the distribution functions. This is especially true with the convolutions performed in the calculation of passage time densities. Many techniques have been used for representing arbitrary distributions – two of the most popular being *phase-type distributions* [106] and *vector-of-moments* methods. These methods suffer from, respectively, exploding representation size under composition, and containing insufficient information to produce accurate answers after large amounts of composition. Attempts to maintain a wholly symbolic representation are similarly hamstrung by space constraints.

As all the distribution manipulations in the algorithm take place in s-space, the distribution representation is linked to the Laplace inversion technique used. The two Laplace transform inversion algorithms which are applied in this thesis are described in Chapter 2. Both work on the same general principle of sampling the transform function $L(s)$ at n points, $s_1, s_2, \ldots, s_n$ and generating values of $f(t)$ at m user-specified t-points $t_1, t_2, \ldots, t_m$. In the Euler inversion case $n = (k + m + 1)$, where k can vary between 15 and 50, depending on the accuracy of the inversion required. In the modified Laguerre case, $n = 400$ and is independent of m (cf. Section 2.3.2).

Whichever Laplace transform inversion technique is employed, it is important to note that calculating $s_i, 1 \leq i \leq n$ and storing all the state holding time distribution transform functions, sampled at these points, will be sufficient to provide a complete inversion. Key to this is the fact that convolution and weighted sum operations do not require any adjustment to the array of domain s-points required. In the case of a convolution, for instance, if $L_1(s)$ and $L_2(s)$ are stored in the form $\{(s_i, L_j(s_i)) \,:\, 1 \leq i \leq n\}$,

for $j = 1, 2$, then the convolution, $L_1(s)L_2(s)$, can be stored using the same size array and using the same list of domain s-values, $\{(s_i, L_1(s_i)L_2(s_i)) \ : \ 1 \leq i \leq n\}$.

Storing the distribution functions in this way has three main advantages. Firstly, the function has constant storage space, independent of the distribution type. Secondly, each distribution has, therefore, the same constant storage requirement even after composition with other distributions. Finally, the function has sufficient information about a distribution to determine the required passage time, and no more.

4.2 The Laplace Transform Method for SMPs

The Laplace transform-based method described in Chapter 3 for the extraction of passage times from Markov models can be extended to the analysis of semi-Markov models. Consider a finite, irreducible, continuous-time semi-Markov process with N states $\{1, 2, \ldots, N\}$. Recalling that $Z(t)$ denotes the state of the SMP at time t ($t \geq 0$) and that $N(t)$ denotes the number of transitions which have occurred by time t, the first passage time from a source state i at time t into a non-empty set of target states $\vec{j}$ is defined as:

$$P_{i\vec{j}}(t) = \inf\{u > 0 : Z(t+u) \in \vec{j}, N(t+u) > N(t), Z(t) = i\}$$

For a stationary time-homogeneous SMP, $P_{i\vec{j}}(t)$ is independent of t:

$$P_{i\vec{j}} = \inf\{u > 0 : Z(u) \in \vec{j}, N(u) > 0, Z(0) = i\} \tag{4.1}$$

$P_{i\vec{j}}$ has an associated probability density function $f_{i\vec{j}}(t)$. In a similar way to Section 3.1, the Laplace transform of $f_{i\vec{j}}(t)$, $L_{i\vec{j}}(s)$, can be computed by means of a first-step analysis. That is, we consider moving from the source state i into the set of its immediate successors $\vec{k}$ and must distinguish between those members of $\vec{k}$ which are target states and those which are not. This calculation can be achieved by solving a set of N linear equations of the form:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} r^*_{ik}(s) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} r^*_{ik}(s) \qquad : \text{for } 1 \leq i \leq N \tag{4.2}$$

where $r^*_{ik}(s)$ is the Laplace-Stieltjes transform (LST) of $R(i,k,t)$ from Section 2.1.3 and is defined by:

$$r^*_{ik}(s) = \int_0^\infty e^{-st}\, dR(i,k,t) \tag{4.3}$$

Eq. 4.2 has a matrix–vector form $\mathbf{Ax} = \mathbf{b}$, where the elements of $\mathbf{A}$ are general functions of the complex variable s. For example, when $\vec{j} = \{1\}$, Eq. 4.2 yields:

$$\begin{pmatrix} 1 & -r^*_{12}(s) & \cdots & -r^*_{1N}(s) \\ 0 & 1-r^*_{22}(s) & \cdots & -r^*_{2N}(s) \\ 0 & -r^*_{32}(s) & \cdots & -r^*_{3N}(s) \\ \vdots & \vdots & \ddots & \vdots \\ 0 & -r^*_{N2}(s) & \cdots & 1-r^*_{NN}(s) \end{pmatrix} \begin{pmatrix} L_{1\vec{j}}(s) \\ L_{2\vec{j}}(s) \\ L_{3\vec{j}}(s) \\ \vdots \\ L_{N\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} r^*_{11}(s) \\ r^*_{21}(s) \\ r^*_{31}(s) \\ \vdots \\ r^*_{N1}(s) \end{pmatrix} \tag{4.4}$$

When there are multiple source states, denoted by the vector $\vec{i}$, the Laplace transform of the passage time density at steady-state is:

$$L_{\vec{i}\vec{j}}(s) = \sum_{k \in \vec{i}} \alpha_k L_{k\vec{j}}(s) \tag{4.5}$$

where the weight α_k is the probability of being in state $k \in \vec{i}$ at the starting instant of the passage. If measuring the system from equilibrium then $\boldsymbol{\alpha}$ is a normalised steady-state vector. That is, if $\boldsymbol{\pi}$ denotes the steady-state vector of the embedded discrete-time Markov chain (DTMC) with one-step transition probability matrix $\mathbf{P} = [p_{ij}, 1 \leq i, j \leq N]$, then α_k is given by:

$$\alpha_k = \begin{cases} \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{if } k \in \vec{i} \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

The row vector with components α_k is denoted by $\boldsymbol{\alpha}$.

4.3 Iterative Passage Time Analysis

In this section, we present an iterative algorithm for generating passage time densities that creates successively better approximations to the SMP passage time quantity $P_{i\vec{j}}$ of Eq. 4.1. We approximate $P_{i\vec{j}}$ as $P^{(r)}_{i\vec{j}}$, for a sufficiently large value of r, which is the

time for r consecutive transitions to occur starting from state i and ending in any of the states in $\vec{j}$. We calculate $P_{i\vec{j}}^{(r)}$ by constructing and then inverting its Laplace transform $L_{i\vec{j}}^{(r)}(s)$.

This iterative method bears a loose resemblance to the uniformization technique described in Section 3.3 which can be used to generate transient state distributions and passage time densities for Markov chains. However, as we are working with semi-Markov systems, there can be no *uniformizing* of the general distributions in the SMP. The general distribution information has to be maintained as precisely as possible throughout the process, which we achieve using the representation technique described in Section 4.1.

4.3.1 Technical Overview

Recall the semi-Markov process $Z(t)$ of Section 2.1.3, where $N(t)$ is the number of state transitions that have taken place by time t. We formally define the rth transition first passage time to be:

$$P_{i\vec{j}}^{(r)} = \inf\{u > 0 : Z(u) \in \vec{j}, 0 < N(u) \leq r, Z(0) = i\} \tag{4.7}$$

which is the time taken to enter a state in $\vec{j}$ for the first time having started in state i at time 0 and having undergone up to r state transitions.

$P_{i\vec{j}}^{(r)}$ is a random variable with associated Laplace transform $L_{i\vec{j}}^{(r)}(s)$. $L_{i\vec{j}}^{(r)}(s)$ is, in turn, the ith component of the vector:

$$\mathbf{L}_{\vec{j}}^{(r)}(s) = \left(L_{1\vec{j}}^{(r)}(s), L_{2\vec{j}}^{(r)}(s), \ldots, L_{N\vec{j}}^{(r)}(s)\right)$$

representing the passage time for terminating in $\vec{j}$ for each possible start state. This vector may be computed as:

$$\mathbf{L}_{\vec{j}}^{(r)}(s) = \mathbf{U}\left(\mathbf{I} + \mathbf{U}' + \mathbf{U}'^{2} + \cdots + \mathbf{U}'^{(r-1)}\right)\mathbf{e}_{\vec{j}} \tag{4.8}$$

where $\mathbf{U}$ is a matrix with elements $u_{pq} = r^*_{pq}(s)$ and $\mathbf{U}'$ is a modified version of $\mathbf{U}$ with elements $u'_{pq} = \delta_{p\notin\vec{j}}\, u_{pq}$, where states in $\vec{j}$ have been made absorbing. Here, $\delta_{p\notin\vec{j}} = 1$

if $p \notin \vec{j}$ and 0 otherwise. The initial multiplication with $\mathbf{U}$ in Eq. 4.8 is included so as to generate cycle times for cases such as $L_{ii}^{(r)}(s)$ which would otherwise register as 0 if $\mathbf{U}'$ were used instead. The column vector $\mathbf{e}_{\vec{j}}$ has entries $e_{k\vec{j}} = \delta_{k\in\vec{j}}$, where $\delta_{k\in\vec{j}} = 1$ if k is a target state ($k \in \vec{j}$) and 0 otherwise.

From Eq. 4.1 and Eq. 4.7:

$$P_{i\vec{j}} = P_{i\vec{j}}^{(\infty)} \quad \text{and thus} \quad L_{i\vec{j}}(s) = L_{i\vec{j}}^{(\infty)}(s)$$

This can be generalised to multiple source states $\vec{i}$ using, for example, the normalised steady-state vector $\boldsymbol{\alpha}$ of Eq. 4.6:

$$\begin{aligned} L_{\vec{i}\vec{j}}^{(r)}(s) &= \boldsymbol{\alpha}\mathbf{L}_{\vec{j}}^{(r)}(s) \\ &= \left(\boldsymbol{\alpha}\mathbf{U} + \boldsymbol{\alpha}\mathbf{U}\mathbf{U}' + \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^2 + \cdots + \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^{(r-1)}\right)\mathbf{e}_{\vec{j}} \\ &= \sum_{k=0}^{r-1} \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^k\,\mathbf{e}_{\vec{j}} \end{aligned} \tag{4.9}$$

The sum of Eq. 4.9 can be computed efficiently using sparse matrix–vector multiplications with a vector accumulator, $\boldsymbol{\mu}_r = \sum_{k=0}^{r} \boldsymbol{\alpha}\mathbf{U}'^k$. At each step, the accumulator (initialised as $\boldsymbol{\mu}_0 = \boldsymbol{\alpha}\mathbf{U}$) is updated as $\boldsymbol{\mu}_{r+1} = \boldsymbol{\alpha}\mathbf{U} + \boldsymbol{\mu}_r\mathbf{U}'$. The worst-case time complexity for this sum is $O(N^2 r)$ versus the $O(N^3)$ of typical matrix inversion techniques. In practice, we typically observe $r << N$ for large N (see Section 4.3.3 below).

4.3.2 Example Passage Time Results

In this section, we display passage time densities produced by our iterative passage time algorithm and validate these results by simulation. Readers are referred to Appendices A.4 and A.5 for full details of the Voting and Web-server models in which these passage times are measured.

Fig. 4.1 shows numerical and simulated (using the combined results from 10 simulations of 1 billion transition firings each) results for the time to complete failure (defined as either all booths have failed or all central servers have failed) in an initially fully

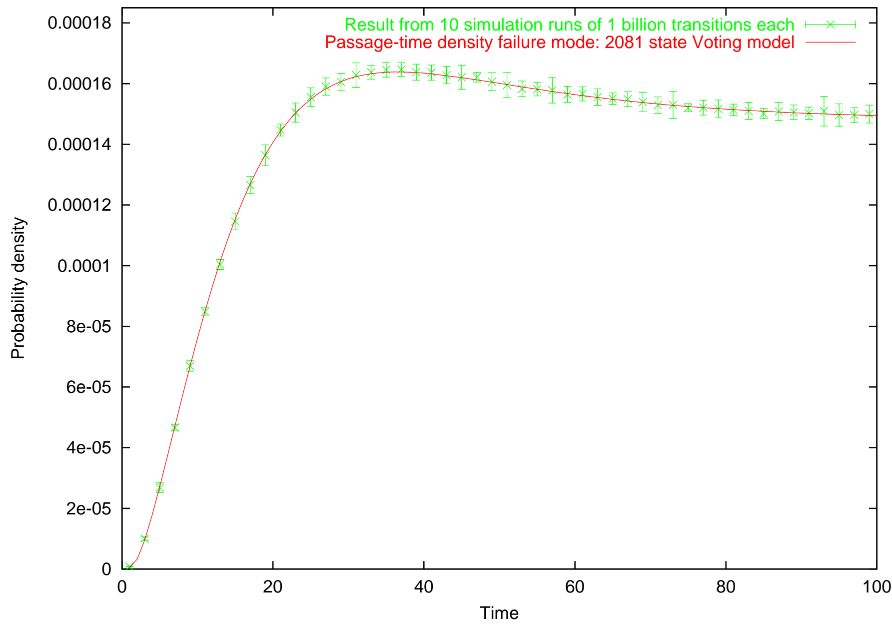

Fig. 4.1. Numerical and simulated (with 95% confidence intervals) density for the failure mode passage in the Voting model system 1 (2 081 states).

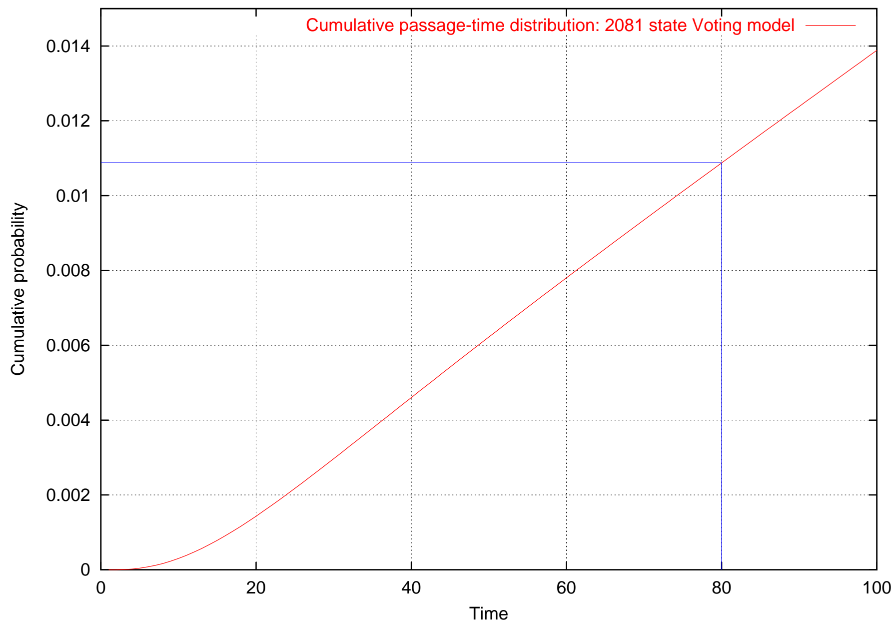

Fig. 4.2. Cumulative distribution function and quantile for the failure mode passage in the Voting model system 1 (2 081 states).

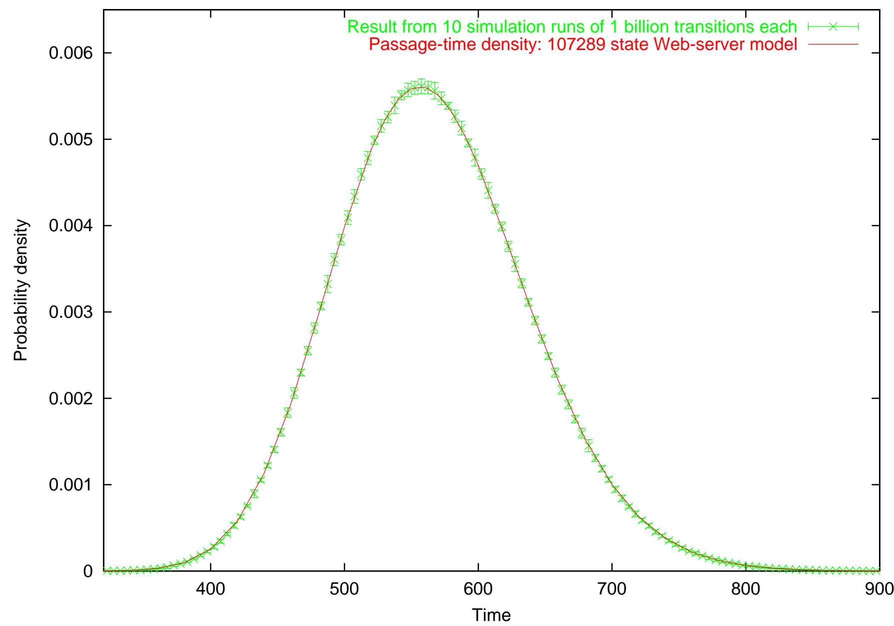

Fig. 4.3. Numerical and simulated (with 95% confidence intervals) density for the time taken to process 45 reads and 22 writes in the Web-server model system 1 (107 289 states).

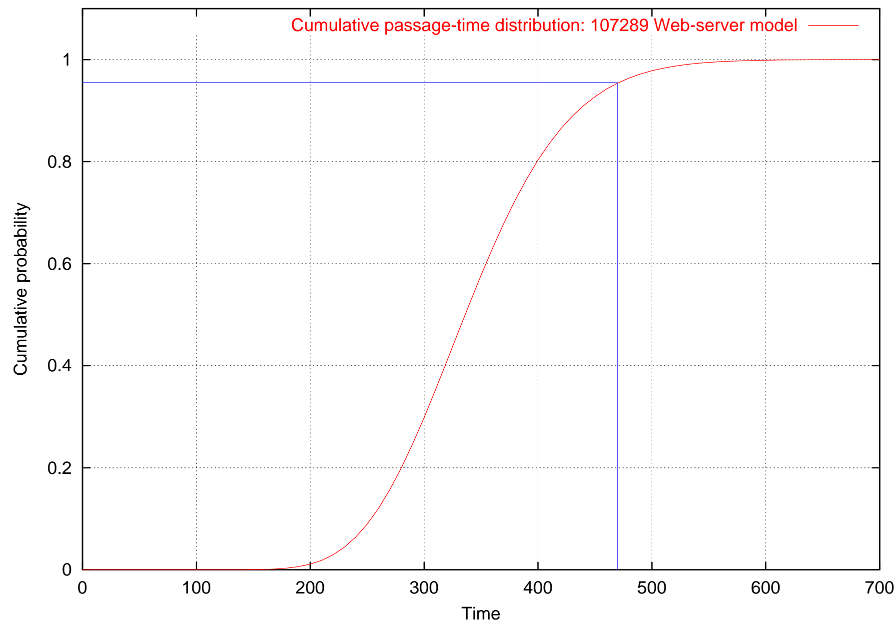

Fig. 4.4. Cumulative distribution function and quantile for the time taken to process 45 reads and 22 writes in the Web-server model system 1 (107 289 states).

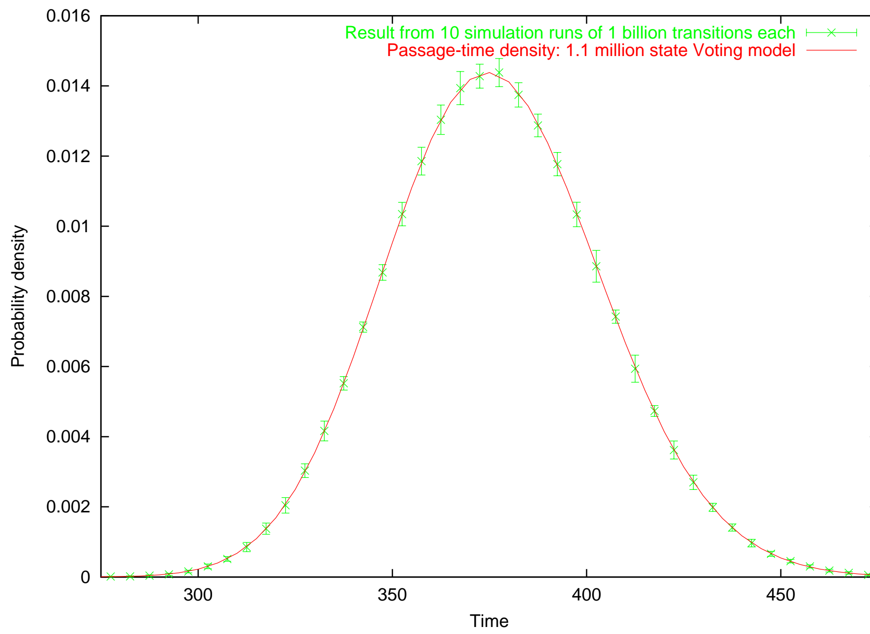

Fig. 4.5. Numerical and simulated (with 95% confidence intervals) density for the time taken to process 175 voters in the Voting model system 7 (1.1 million states).

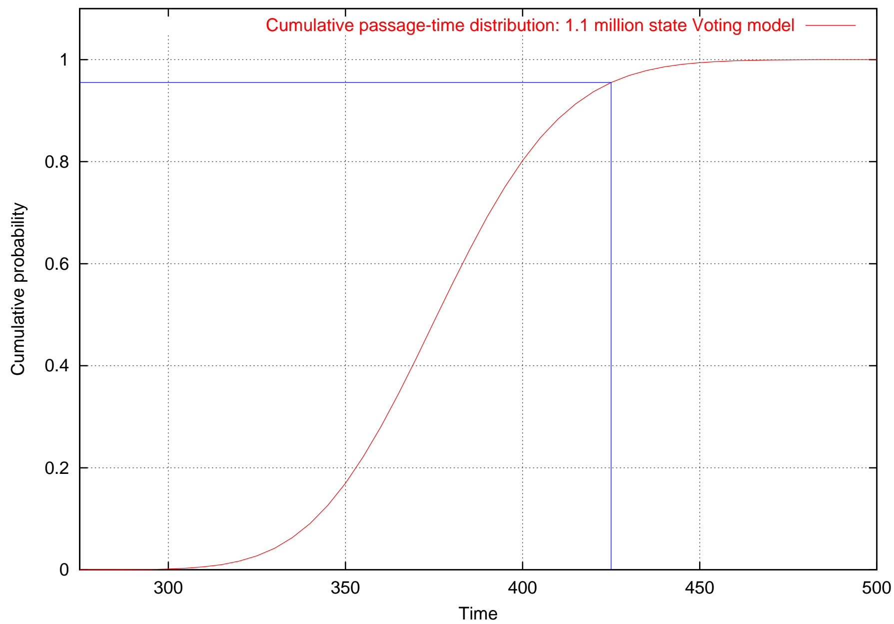

Fig. 4.6. Cumulative distribution function and quantile for the time taken to process 175 voters in the Voting model system 7 (1.1 million states).

operational Voting system. It is produced for a small system ($2\,081$ states) as the probabilities for the larger systems were so small that the simulator was not able to register any meaningful distribution for the quantity without using rare-event techniques. As we wanted to validate the passage time algorithm, we reduced the number of states so that the simulator would register a density. Examining very-low-probability events is an excellent example of where analytical techniques outperform simulations that would take many hours or even days to complete.

Fig. 4.2 shows a cumulative distribution for the same passage as Fig. 4.1 (easily obtained by inverting $L_{\vec{i}\vec{j}}(s)/s$ from cached values of $L_{\vec{i}\vec{j}}(s)$). It allows us to extract response time quantiles, for instance:

$$\mathbb{P}(\text{either all the booths or all the servers fail in system 1 in under 80 seconds}) = 0.0109$$

Fig. 4.3 shows the density of the time taken to process 45 reads and 22 writes in system 1 of the Web-server model ($107\,289$ states). This corresponds to the movement of 45 tokens from p_1 to p_8 and 22 tokens from p_2 to p_9. The graph shows results computed by both the iterative technique and the combined results from 10 simulations of 1 billion transition firings each. The close agreement provides mutual validation of the analytical method, with its numerical approximation, and the simulation.

Fig. 4.4 shows the cumulative distribution for the same passage as Fig. 4.3. An example response time quantile for this measure would be:

$$\mathbb{P}(\text{all reads and all writes are processed in under 470 seconds}) = 0.954$$

Fig. 4.5 shows the density of the time taken for the passage of 175 voters from place p_1 to p_2 in system 7 of the Voting model ($1\,140\,050$ states) starting from when all servers are operational. The results presented are those computed by the iterative technique and the combined results from 10 simulations of 1 billion transition firings each. As with the previous example, the close agreement observed provides mutual validation of the analytical method and the simulation.

Fig. 4.6 shows a cumulative distribution for the same passage as Fig. 4.5. We can extract response time quantiles from it, for instance:

$$\mathbb{P}(\text{system 5 processes 175 voters in under 425 seconds}) = 0.955$$

4.3.3 Practical Convergence of the Iterative Passage Time Algorithm

In practice, convergence of the sum $L_{\vec{i}\vec{j}}^{(r)}(s) = \sum_{k=0}^{r-1} \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^{k}$ can be said to have occurred if, for a particular r and s-point:

$$|\mathrm{Re}(L_{\vec{i}\vec{j}}^{(r+1)}(s) - L_{\vec{i}\vec{j}}^{(r)}(s))| < \varepsilon \quad \text{and} \quad |\mathrm{Im}(L_{\vec{i}\vec{j}}^{(r+1)}(s) - L_{\vec{i}\vec{j}}^{(r)}(s))| < \varepsilon \qquad (4.10)$$

where ε is chosen to be a suitably small value, say $\varepsilon = 10^{-16}$. Empirical observations on the convergence behaviour of this technique (i.e. the order of r) are presented below.

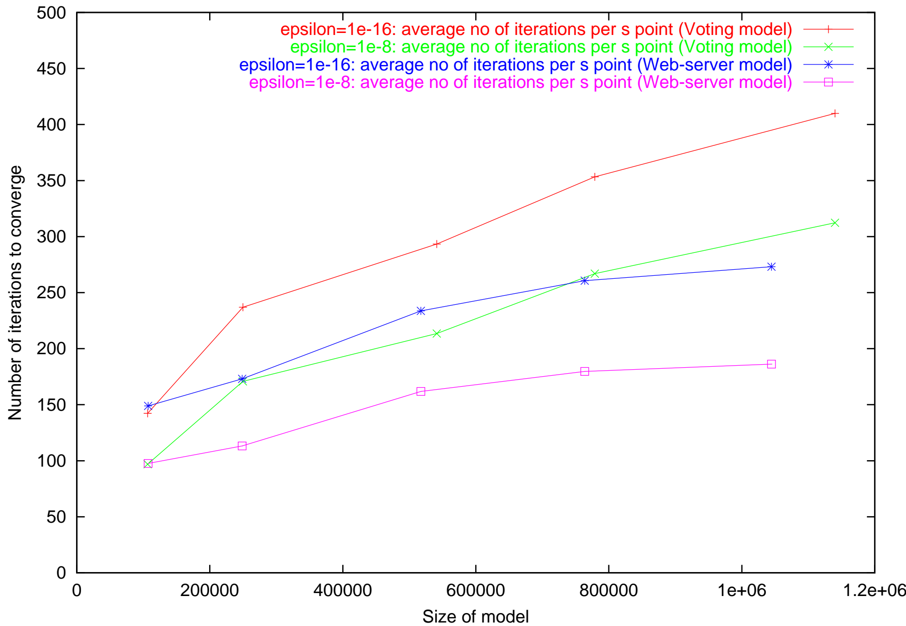

Fig. 4.7. Average number of iterations to converge per s point for two different values of ε over a range of model sizes for the iterative passage time algorithm.

Fig. 4.7 shows the average number of iterations the algorithm takes to converge per s-point for the Voting and Web-server models (see Appendices A.4 and A.5 for full details) for two different values of ε (10^{-8} and 10^{-16}). It is noted that the number of iterations required for convergence as the model size grows is sub-linear; that is, as the model size doubles the number of iterations less than doubles. This suggests the algorithm has good scalability properties.

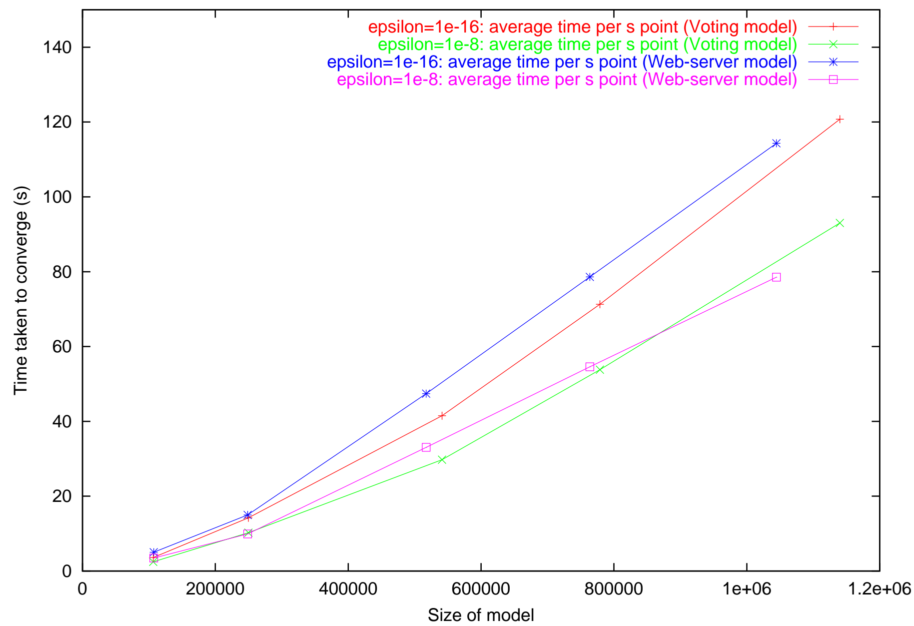

Fig. 4.8. Average time to convergence per s point for two different values of ε over a range of model sizes for the iterative passage time algorithm.

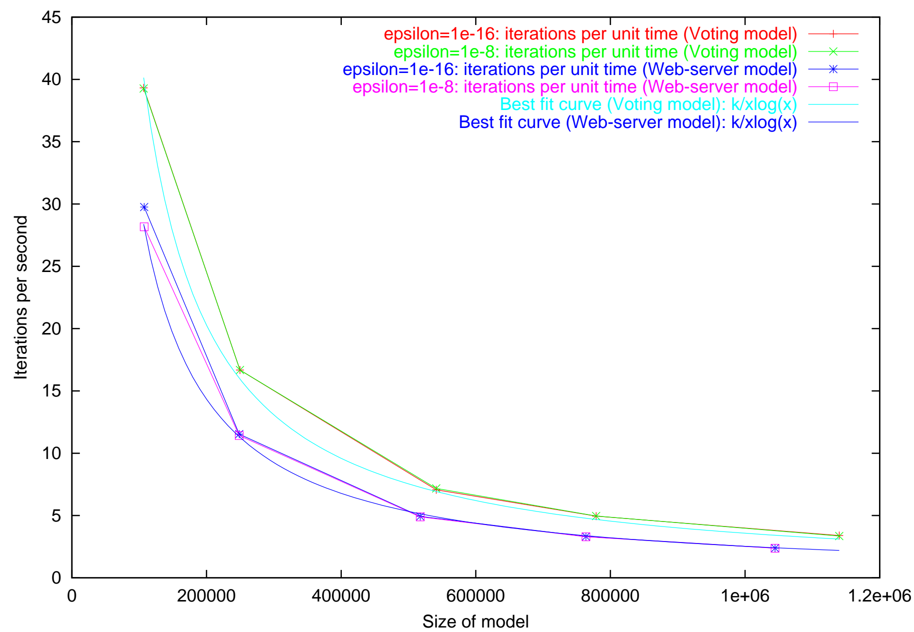

Fig. 4.9. Average number of iterations per unit time over a range of model sizes for the iterative passage time algorithm.

Fig. 4.8 shows the average amount of time to convergence per s-point, while Fig. 4.9 shows how the number of iterations per unit time decreases as model size increases. The curves are almost identical for both values of ε, suggesting that the time spent per iteration remains constant, irrespective of the number of iterations performed. The rate of computation (iterations per unit time) is $O\big(1/(N\log(N))\big)$ for system size N. This gives a time per iteration of $O\big(N\log(N)\big)$, suggesting an overall practical complexity of better than $O\big(N^2\log(N)\big)$ (given the better than $O(N)$ result for the number of iterations required).

4.4 Iterative Transient Analysis

Another important modelling result is the transient state distribution $\pi_{ij}(t)$ of a stochastic process:

$$\pi_{ij}(t) = \mathbb{P}(Z(t) = j \mid Z(0) = i)$$

From Pyke's seminal paper on SMPs [110], we have the following relationship between passage time densities and transient state distributions, in Laplace form:

$$\pi^*_{ij}(s) = \frac{1}{s}\frac{1-h^*_i(s)}{1-L_{ii}(s)} \quad \text{if } i=j, \qquad \pi^*_{ij}(s) = L_{ij}(s)\pi^*_{jj}(s) \quad \text{if } i \neq j$$

where $\pi^*_{ij}(s)$ is the Laplace transform of $\pi_{ij}(t)$ and $h^*_i(s) = \sum_k r^*_{ik}(s)$ is the LST of the sojourn time distribution in state i. For multiple target states, this becomes:

$$\pi^*_{i\vec{j}}(s) = \sum_{k\in\vec{j}} \pi^*_{ik}(s)$$

However, to construct $\pi^*_{i\vec{j}}(s)$ directly using this translation is computationally expensive: for a vector of target states $\vec{j}$, we need $2|\vec{j}|-1$ passage time quantities, $L_{ik}(s)$, which in turn require the solution of $|\vec{j}|$ linear systems of the form of Eq. 4.4. This motivates our development of a new transient state distribution formula for multiple target states in semi-Markov processes which requires the solution of only one system of linear equations per s-value.

From Pyke's formula for the transient state distribution between two states [110, Eq.

(3.2)], we can derive:

$$\pi_{ij}(t) = \delta_{ij}\overline{F}_i(t) + \sum_{k=1}^{N}\int_0^t R(i,k,t-\tau)\,\pi_{kj}(\tau)\,d\tau$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise, and $\overline{F}_i(t)$ is the reliability function of the sojourn time distribution in state i, i.e. the probability that the system has not left state i after t time units. $R(i,k,t-\tau)$ is the probability that a transition from state i to an adjacent state k occurs in time $t-\tau$ and $\pi_{kj}(\tau)$ is the probability of being in state j having left state k after a further time τ.

Transforming this convolution into the Laplace domain and generalising to multiple target states, $\vec{j}$, we obtain:

$$\pi^*_{i\vec{j}}(s) = \delta_{i\in\vec{j}}\overline{F}^*_i(s) + \sum_{k=1}^{N} r^*_{ik}(s)\pi^*_{k\vec{j}}(s) \tag{4.11}$$

Here, $\delta_{i\in\vec{j}} = 1$ if $i \in \vec{j}$ and 0 otherwise. The Laplace transform of the reliability function $\overline{F}^*_i(s)$ is generated from $h^*_i(s)$ as:

$$\overline{F}^*_i(s) = \frac{1-h^*_i(s)}{s}$$

Eq. 4.11 can be written in matrix–vector form; for example, when $\vec{j} = \{1,3\}$, we have:

$$\begin{pmatrix} 1-r^*_{11}(s) & -r^*_{12}(s) & \cdots & -r^*_{1N}(s) \\ -r^*_{21}(s) & 1-r^*_{22}(s) & \cdots & -r^*_{2N}(s) \\ -r^*_{31}(s) & -r^*_{32}(s) & \cdots & -r^*_{3N}(s) \\ \vdots & \vdots & \ddots & \vdots \\ -r^*_{N2}(s) & -r^*_{N2}(s) & \cdots & 1-r^*_{NN}(s) \end{pmatrix} \begin{pmatrix} \pi^*_{1\vec{j}}(s) \\ \pi^*_{2\vec{j}}(s) \\ \pi^*_{3\vec{j}}(s) \\ \vdots \\ \pi^*_{N\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} \overline{F}^*_1(s) \\ 0 \\ \overline{F}^*_3(s) \\ \vdots \\ 0 \end{pmatrix} \tag{4.12}$$

Again for multiple source states with initial distribution $\boldsymbol{\alpha}$, the Laplace transform of the transient function is:

$$\pi^*_{\vec{i}\vec{j}}(s) = \sum_{k\in\vec{i}} \alpha_k \pi^*_{k\vec{j}}(s)$$

4.4.1 Technical Overview

Our iterative transient state distribution generation technique builds on the passage time computation technique of Section 4.3. We aim to calculate $\pi_{\vec{i}\vec{j}}(t)$, that is the

probability of being in any of the states of $\vec{j}$ at time t having started in state i at time $t = 0$. We approximate this transient state distribution by constructing and then inverting $\pi^{(r)}_{i\vec{j}}(s)$, which is the rth iterative approximation to the Laplace transform of the transient state distribution function, for a sufficiently large value of r.

We note that Eq. 4.12 can be written as:

$$(\mathbf{I} - \mathbf{U})\,\boldsymbol{\pi}_{\vec{j}}(s) = \mathbf{v} \tag{4.13}$$

where matrix $\mathbf{U}$ has elements $u_{pq} = r^*_{pq}(s)$ and column vector $\mathbf{v}$ has elements $v_i = \delta_{i\in\vec{j}}\overline{F}^*_i(s)$. The vector $\boldsymbol{\pi}_{\vec{j}}(s)$ has elements $\pi_{i\vec{j}}(s)$:

$$\boldsymbol{\pi}_{\vec{j}}(s) = \Big(\pi_{1\vec{j}}(s), \pi_{2\vec{j}}(s), \ldots, \pi_{N\vec{j}}(s)\Big)$$

Eq. 4.13 can be rewritten (see [30] for the proof that $(\mathbf{I} - \mathbf{U})$ is invertible) as:

$$\begin{aligned}\boldsymbol{\pi}_{\vec{j}}(s) &= (\mathbf{I} - \mathbf{U})^{-1}\,\mathbf{v} \\ &= \Big(\mathbf{I} + \mathbf{U} + \mathbf{U}^2 + \mathbf{U}^3 + \cdots\Big)\,\mathbf{v}\end{aligned}$$

This infinite summation may be approximated as:

$$\boldsymbol{\pi}_{\vec{j}}(s) \simeq \boldsymbol{\pi}^{(r)}_{\vec{j}}(s) = \Big(\mathbf{I} + \mathbf{U} + \mathbf{U}^2 + \cdots + \mathbf{U}^r\Big)\,\mathbf{v}$$

for a suitable value of r such that the approximation is good. See Section 4.4.3 below for observations regarding typical values of r.

Note that instead of using an absorbing transition matrix as in the passage time scheme, the transient method makes use of the unmodified transition matrix $\mathbf{U}$. This reflects the fact that the transient state distribution accumulates probability from all passages through the system and not just the first one.

Finally, as before, the technique can be generalised to multiple start states by employing an initial row vector $\boldsymbol{\alpha}$, where α_i is the probability of being in state i at time 0:

$$\pi^{(r)}_{\vec{i}\vec{j}}(s) = \boldsymbol{\alpha}\Big(\mathbf{I} + \mathbf{U} + \mathbf{U}^2 + \cdots + \mathbf{U}^r\Big)\,\mathbf{v}$$

Having calculated $\pi^{(r)}_{\vec{i}\vec{j}}(s)$ in this manner, the same numerical inversion techniques which are used in passage time analysis can be employed to compute $\pi_{\vec{i}\vec{j}}(t)$.

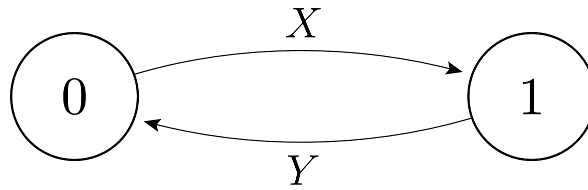

Fig. 4.10. A simple two-state semi-Markov process.

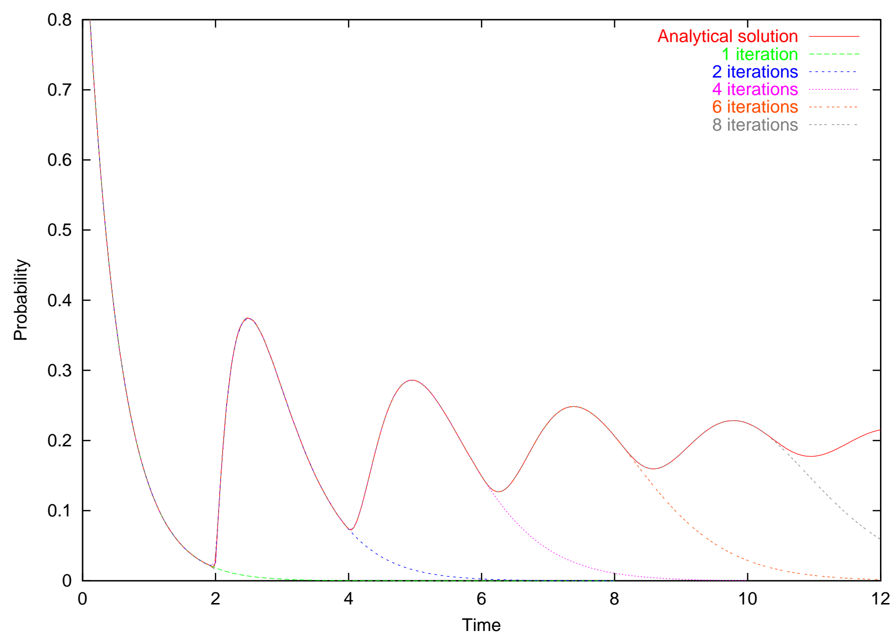

Fig. 4.11. Example iterations towards a transient state distribution in a system with successive exponential and deterministic transitions.

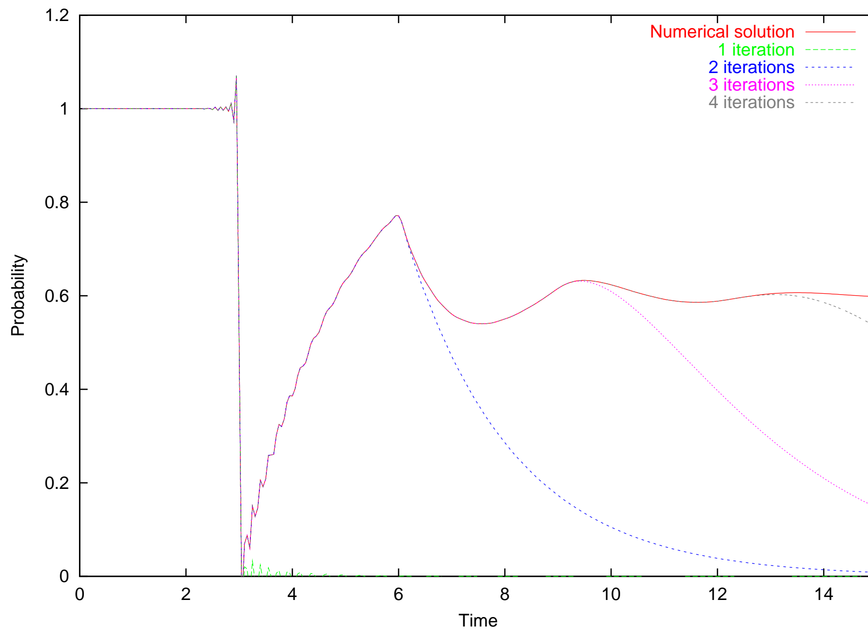

Fig. 4.12. Example iterations towards a transient state distribution in a system with successive deterministic and exponential transitions.

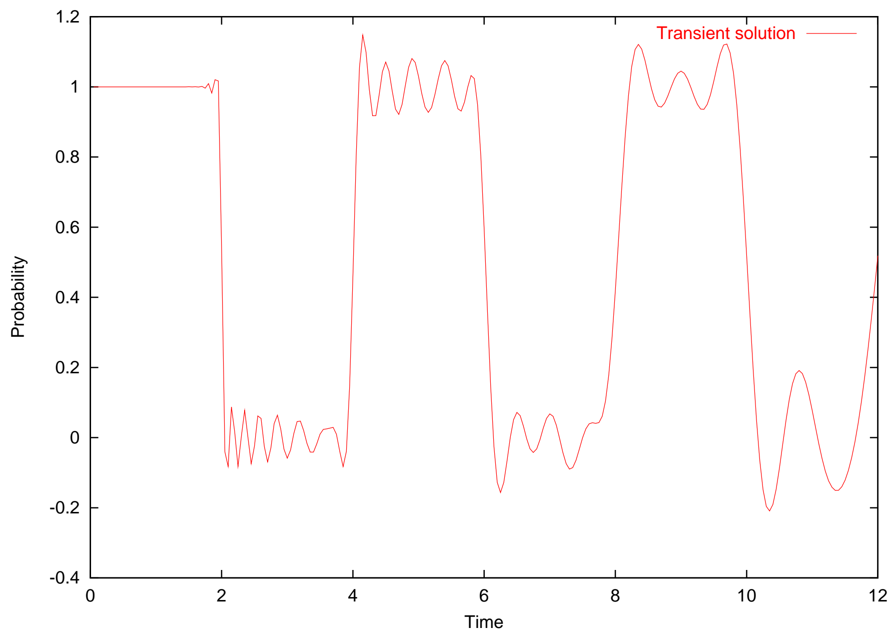

Fig. 4.13. Where numerical inversion performs badly: transient state distribution in a system with two deterministic transitions.

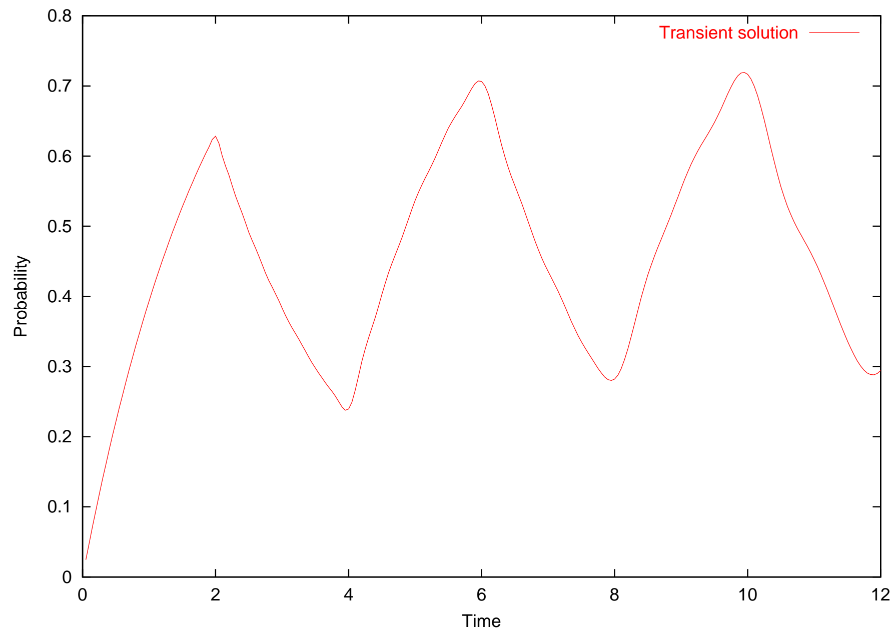

Fig. 4.14. The effect of adding randomness: transient state distribution of the two deterministic transitions system with a initial exponential transition added.

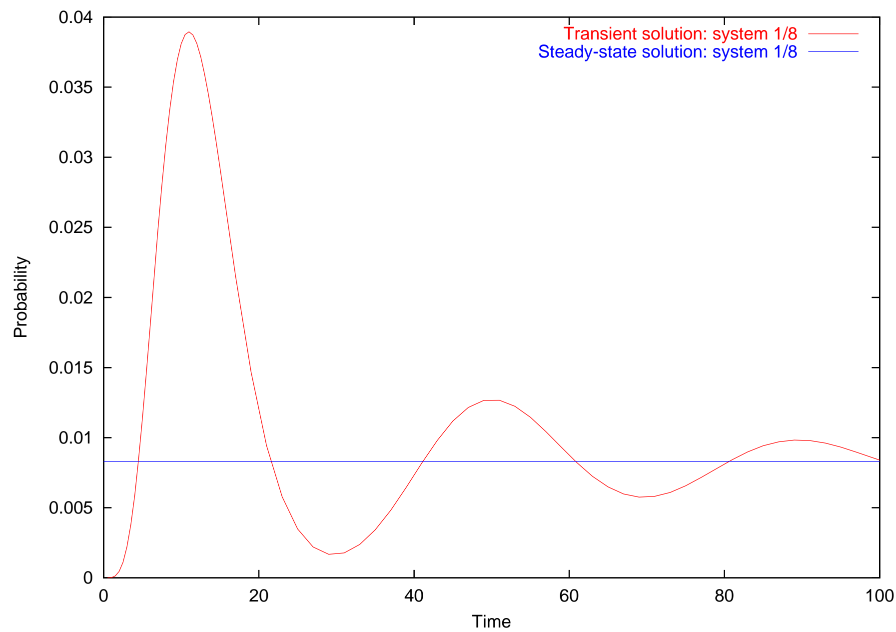

Fig. 4.15. Transient and steady-state values in system 1, for the transit of 5 voters from the initial marking to place p_2.

4.4.2 Example Transient Results

We demonstrate our iterative transient technique on two models: the small two-state example shown in Fig. 4.10 and the Voting model described in Appendix A.4.

For the two-state example, Fig. 4.11 shows a transient state distribution $\pi_{00}(t)$, that is the probability of being in state 0, having started in state 0, at time t. The distributions of the transitions are $X \sim exp(2)$ and $Y \sim det(2)$. The discontinuities in the derivative from the deterministic transition can clearly be made out at points $t = 2, 4$ and in fact also exist at $t = 6, 8, 10, \ldots$. Also shown on the graph are up to 8 iterations of the algorithm which exhibit increasing accuracy in approximating the transient curve.

Fig. 4.12 shows the transient state distribution $\pi_{00}(t)$ for the two state system with $X \sim det(3)$ and $Y \sim exp(0.5)$. The graph clearly shows the system remaining in state 0 for the initial 3 time units, as dictated by the out-going deterministic transition. The perturbations in the graph observed around $t = 3$ are generated by numerical instabilities (Gibb's Phenomena) in the Laplace inversion algorithm [3]. Also shown on the graph are 4 iterations of the algorithm which exhibit increasing accuracy in approximating the transient curve, as before.

We also use the two state system of Fig. 4.10 to highlight when numerical Laplace transform inversion does not perform well and how such problems can be avoided. Fig. 4.13 shows the transient probability of being in state 0 having started in state 0 when both X and Y are $det(2)$ transitions. We would expect to see the probability equalling 1 for $0 < t < 2$, $4 < t < 6$ and so forth, and 0 at $2 < t < 4$, $6 < t < 8$ and so on, but the numerically computed result becomes increasingly unstable as t increases. This is because discontinuities in $f(t)$ and its derivatives result in instabilities in the numerical inversion. Even the Euler algorithm (which was used to produce these results) performs badly when inverting entirely deterministic probability distributions. This example, with two such transitions and no source of randomness, is the worst case we could expect to deal with.

The presence of a small amount of randomness is, however, enough to remove this instability. We modify the two state system by adding a new state with a single $exp(0.5)$

transition into state 0 and calculate the transient probability of being in state 0 having started in the newly added predecessor state. There is no transition from state 0 back to the new state, so the $exp(0.5)$ transition fires only once. The resulting transient state distribution is shown in Fig. 4.14. Note that the numerical instability has disappeared. This demonstrates that only a small amount of randomness in the model can be sufficient for numerical inversion to be applied successfully.

For the Voting model, Fig. 4.15 shows the transient state distribution for the transit of five voters from place p_1 to p_2 in system 1. As expected, the distribution tends towards its steady-state value as $t \to \infty$.

4.4.3 Practical Convergence of the Iterative Transient Algorithm

As in the iterative passage time algorithm, convergence of the sum $\pi^{(r)}_{\vec{i}\vec{j}}(s)$ is said to have occurred if, for a particular r and s-point:

$$|\mathrm{Re}(\pi^{(r+1)}_{\vec{i}\vec{j}}(s) - \pi^{(r)}_{\vec{i}\vec{j}}(s))| < \varepsilon \quad \text{and} \quad |\mathrm{Im}(\pi^{(r+1)}_{\vec{i}\vec{j}}(s) - \pi^{(r)}_{\vec{i}\vec{j}}(s))| < \varepsilon \tag{4.14}$$

where ε is chosen to be a suitably small value such as 10^{-16}.

Fig. 4.16 shows the average number of iterations the algorithm takes to converge per s-point for the Voting model for two different values of ε (10^{-8} and 10^{-16}). It is interesting to note that, in contrast to the iterative passage time algorithm, the number of iterations until convergence is achieved appears to remain constant as the number of states increases.

Fig. 4.17 shows the average amount of time to convergence per s-point, while Fig. 4.18 shows how the number of iterations per unit time decreases as model size increases. As with the iterative passage time algorithm, we observe an overall practical complexity of better than $O\big(N^2 \log(N)\big)$.

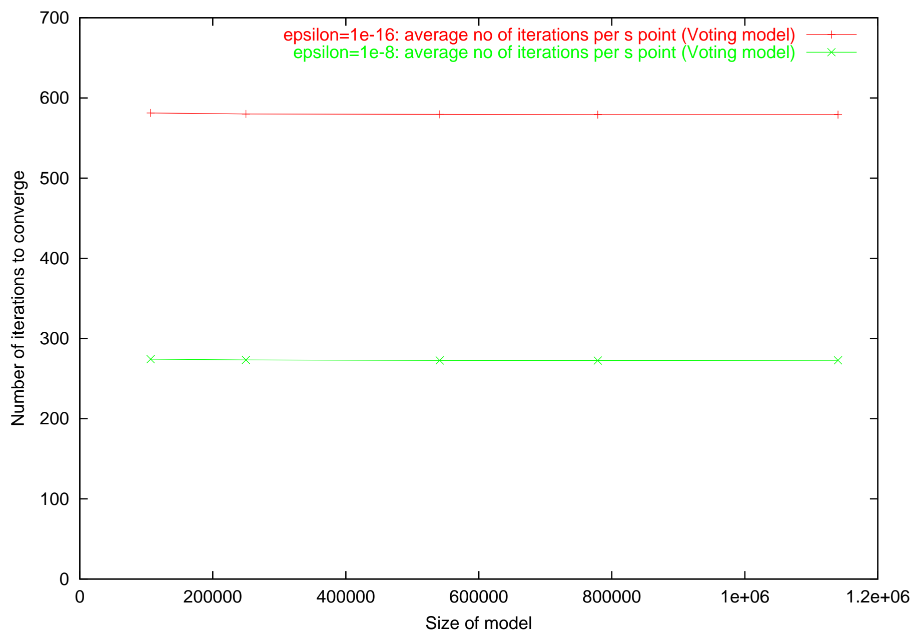

Fig. 4.16. Average number of iterations to converge per s point for two different values of ε over a range of model sizes for the iterative transient algorithm.

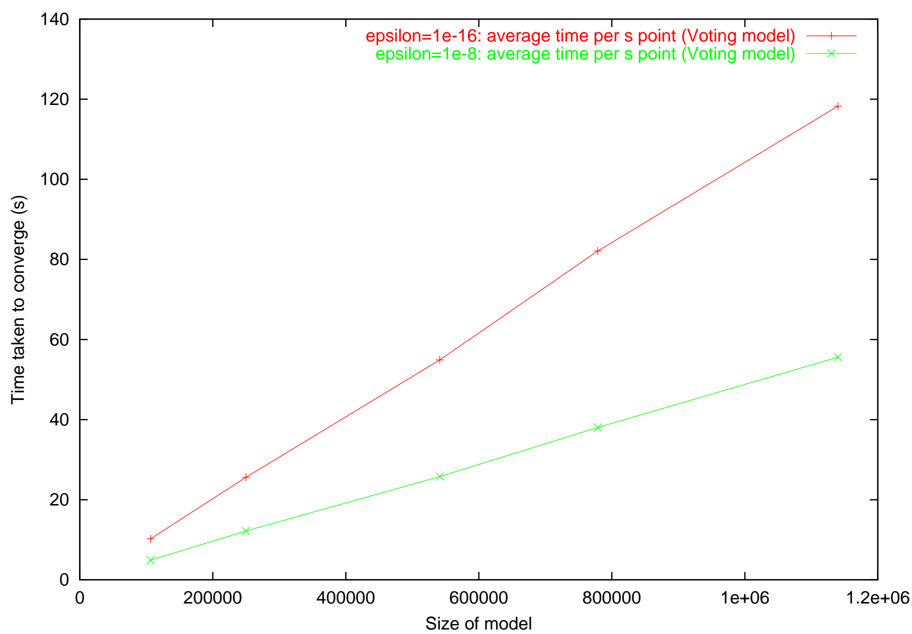

Fig. 4.17. Average time to convergence per s point for two different values of ε over a range of model sizes for the iterative transient algorithm.

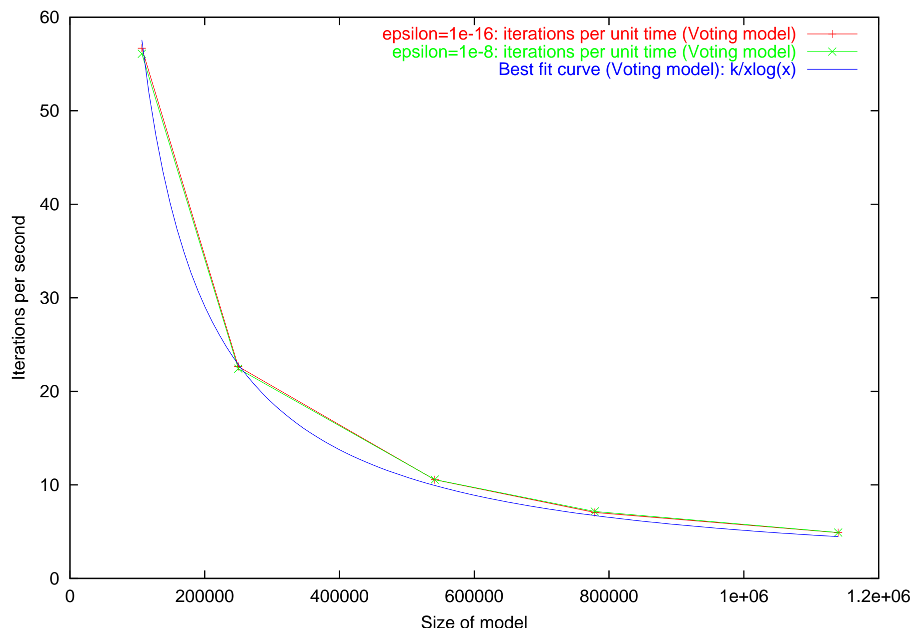

Fig. 4.18. Average number of iterations per unit time over a range of model sizes for the iterative transient algorithm.

4.5 Estimation of Passage Time Densities and Distributions From Their Moments

As well as calculating a full passage time density in a semi-Markov model using the iterative algorithm, it is also possible to approximate it from its moments. The technique described in this section is an extension of the procedure described for calculating moments in Markov systems in Section 3.6. Once the first four moments have been calculated in the manner described below, the corresponding density or distribution function can be approximated using the technique described in Section 3.6.3.

As in Chapter 3, we define the nth raw moment of the Laplace transform of the passage time density $L_{i\vec{j}}(s)$ as:

$$M_{i\vec{j}}(n) = (-1)^n \left. \frac{d^n L_{i\vec{j}}(s)}{ds^n} \right|_{s=0}$$

That is to say, the nth moment is calculated by differentiating $L_{i\vec{j}}(s)$ n times and evaluating at $s = 0$. As we are dealing with semi-Markov processes, we must also deal with generally-distributed state sojourn times. We therefore define the quantity

$m_{ik}(n)$ for an SMP as [70]:

$$m_{ik}(n) = (-1)^n \left. \frac{d^n r^*_{ik}(s)}{ds^n} \right|_{s=0}$$

4.5.1 Moment Calculation

The general formula for calculating the nth moment of passage time for a semi-Markov model is stated without proof in [70]:

$$M_{i\vec{j}}(n) = \sum_{k \notin \vec{j}} \sum_{r=0}^{n} \binom{n}{r} m_{ik}(r) M_{k\vec{j}}(n-r) + \sum_{k \in \vec{j}} m_{ik}(n) \tag{4.15}$$

for $i \notin \vec{j}$ and $M_{i\vec{j}}(n) = 0$ for $i \in \vec{j}$. Also, $M_{i\vec{j}}(0) = 1$ and $p_{ik} = r^*_{ik}(0) \equiv m_{ik}(0)$. We will prove that this general formula holds by induction.

For a semi-Markov model, we calculate the Laplace transform of the density of the passage time between states i and states $\vec{j}$ by solving a system of linear equations of the form:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} r^*_{ik}(s) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} r^*_{ik}(s) \qquad : \text{for } 1 \leq i \leq N \tag{4.16}$$

where r^*_{ik} is as defined in Eq. 4.3. We will now show that differentiating Eq. 4.16 n times gives:

$$L^{(n)'}_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} \sum_{r=0}^{n} \binom{n}{r} r^{*(r)'}_{ik}(s) L^{(n-r)'}_{k\vec{j}}(s) + \sum_{k \in \vec{j}} r^{*(n)'}_{ik}(s) \tag{4.17}$$

for any integer value of n.

Base case Differentiating Eq. 4.16 once requires the use of the product rule, and this yields:

$$L'_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} \Big(r^*_{ik}(s) L'_{k\vec{j}}(s) + r'^*_{ik}(s) L_{k\vec{j}}(s) \Big) + \sum_{k \in \vec{j}} r'^*_{ik}(s)$$

which is Eq. 4.17 for $n = 1$.

Inductive step Given that Eq. 4.17 holds, we must show that differentiating it once gives Eq. 4.17 for the $(n+1)$th differential. Applying the product rule we have:

$$
\begin{aligned}
L_{i\vec{j}}^{(n+1)'}(s) &= \sum_{k \notin \vec{j}} \sum_{r=0}^{n} \binom{n}{r} \left(r_{ik}^{*(r+1)'}(s) L_{k\vec{j}}^{(n-r)'}(s) + r_{ik}^{*(r)'}(s) L_{k\vec{j}}^{(n-r+1)'}(s) \right) \\
&\quad + \sum_{k \in \vec{j}} r_{ik}^{*(n+1)'}(s)
\end{aligned}
\tag{4.18}
$$

The first summation over $k \notin \vec{j}$ requires special attention. Expanding it yields:

$$
\begin{aligned}
&\sum_{r=0}^{n} \binom{n}{r} \left(r_{ik}^{*(r+1)'}(s) L_{k\vec{j}}^{(n-r)'}(s) + r_{ik}^{*(r)'}(s) L_{k\vec{j}}^{(n-r+1)'}(s) \right) = \\
&\qquad \binom{n}{0} \left(r_{ik}^{*(1)'}(s) L_{k\vec{j}}^{(n)'}(s) + r_{ik}^{*(0)'}(s) L_{k\vec{j}}^{(n+1)'}(s) \right) \\
&\qquad + \binom{n}{1} \left(r_{ik}^{*(2)'}(s) L_{k\vec{j}}^{(n-1)'}(s) + r_{ik}^{*(1)'}(s) L_{k\vec{j}}^{(n)'}(s) \right) + \ldots \\
&\qquad + \binom{n}{n-1} \left(r_{ik}^{*(n)'}(s) L_{k\vec{j}}^{(1)'}(s) + r_{ik}^{*(n-1)'}(s) L_{k\vec{j}}^{(2)'}(s) \right) \\
&\qquad + \binom{n}{n} \left(r_{ik}^{*(n+1)'}(s) L_{k\vec{j}}^{(0)'}(s) + r_{ik}^{*(n)'}(s) L_{k\vec{j}}^{(1)'}(s) \right)
\end{aligned}
$$

Collecting like-terms on the right-hand side gives:

$$
\begin{aligned}
&\sum_{r=0}^{n} \binom{n}{r} \left(r_{ik}^{*(r+1)'}(s) L_{k\vec{j}}^{(n-r)'}(s) + r_{ik}^{*(r)'}(s) L_{k\vec{j}}^{(n-r+1)'}(s) \right) = \\
&\qquad \binom{n}{0} \left(r_{ik}^{*(0)'}(s) L_{k\vec{j}}^{(n+1)'}(s) \right) + \left(\binom{n}{0} + \binom{n}{1} \right) \left(r_{ik}^{*(1)'}(s) L_{k\vec{j}}^{(n)'}(s) \right) \\
&\qquad + \left(\binom{n}{1} + \binom{n}{2} \right) \left(r_{ik}^{*(2)'}(s) L_{k\vec{j}}^{(n-1)'}(s) \right) + \ldots \\
&\qquad + \left(\binom{n}{n-1} + \binom{n}{n} \right) \left(r_{ik}^{*(n)'}(s) L_{k\vec{j}}^{(1)'}(s) \right) \\
&\qquad + \binom{n}{n} \left(r_{ik}^{*(n+1)'}(s) L_{k\vec{j}}^{(0)'}(s) \right)
\end{aligned}
$$

Using the equivalence:

$$
\binom{n}{r} + \binom{n}{r-1} = \binom{n+1}{r}
$$

this can be rewritten as:

$$\sum_{r=0}^{n}\binom{n}{r}\left(r_{ik}^{*(r+1)'}(s)L_{k\vec{j}}^{(n-r)'}(s)+r_{ik}^{*(r)'}(s)L_{k\vec{j}}^{(n-r+1)'}(s)\right)=$$

$$\binom{n+1}{0}\left(r_{ik}^{*(0)'}(s)L_{k\vec{j}}^{(n+1)'}(s)\right)+\binom{n+1}{1}\left(r_{ik}^{*(1)'}(s)L_{k\vec{j}}^{(n)'}(s)\right)$$
$$+\binom{n+1}{2}\left(r_{ik}^{*(2)'}(s)L_{k\vec{j}}^{(n-1)'}(s)\right)+\ldots+\binom{n+1}{n}\left(r_{ik}^{*(n)'}(s)L_{k\vec{j}}^{(1)'}(s)\right)$$
$$+\binom{n+1}{n+1}\left(r_{ik}^{*(n+1)'}(s)L_{k\vec{j}}^{(0)'}(s)\right)$$
$$=\sum_{r=0}^{n+1}\binom{n+1}{r}r_{ik}^{*(r)'}(s)L_{k\vec{j}}^{(n-r+1)'}(s)$$

Substituting back into Eq. 4.18 gives:

$$L_{i\vec{j}}^{(n+1)'}(s)=\sum_{k\notin\vec{j}}\sum_{r=0}^{n+1}\binom{n+1}{r}r_{ik}^{*(r)'}(s)L_{k\vec{j}}^{(n-r+1)'}(s)+\sum_{k\in\vec{j}}r_{ik}^{*(n+1)'}(s)$$

as required.

We have therefore proved that Eq. 4.17 holds for all integer values of $n \geq 1$. Evaluating Eq. 4.17 at $s=0$ now yields Eq. 4.15 as required.

4.5.2 Example Results

A simple SM-SPN model is shown in Fig. 4.19. We demonstrate the estimation of a passage time in an SMP from its moments on the passage of 20 tokens from p_1 to p_0 in this model. The result produced by the iterative technique of Section 4.3 and an approximation produced by applying the GLD method described in Section 3.6.3 to moments calculated using the method of Section 4.5.1 can be seen in Fig. 4.20. We notice good agreement between the exact result and the approximation. The corresponding cumulative distribution function (in the case of the GLD method produced by numerical integration of the probability density function but computed directly for the exact passage times) can also be seen in Fig. 4.21. Again, the agreement is excellent.

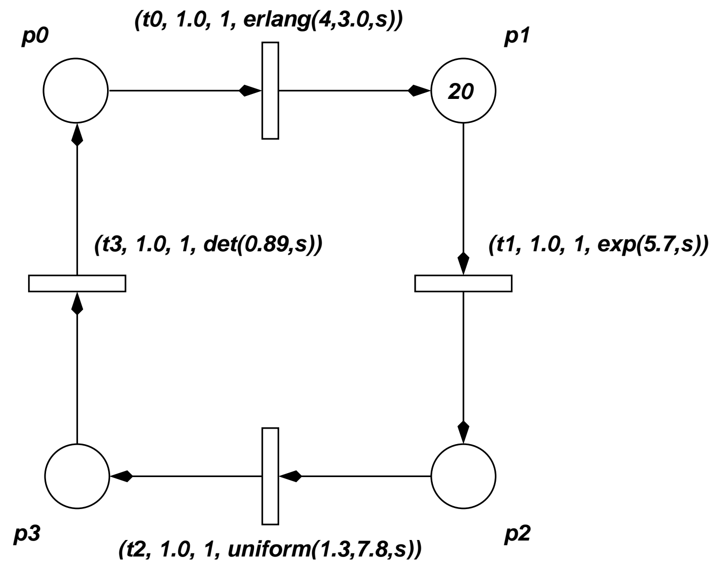

Fig. 4.19. A simple four-state semi-Markov model (SMFour).

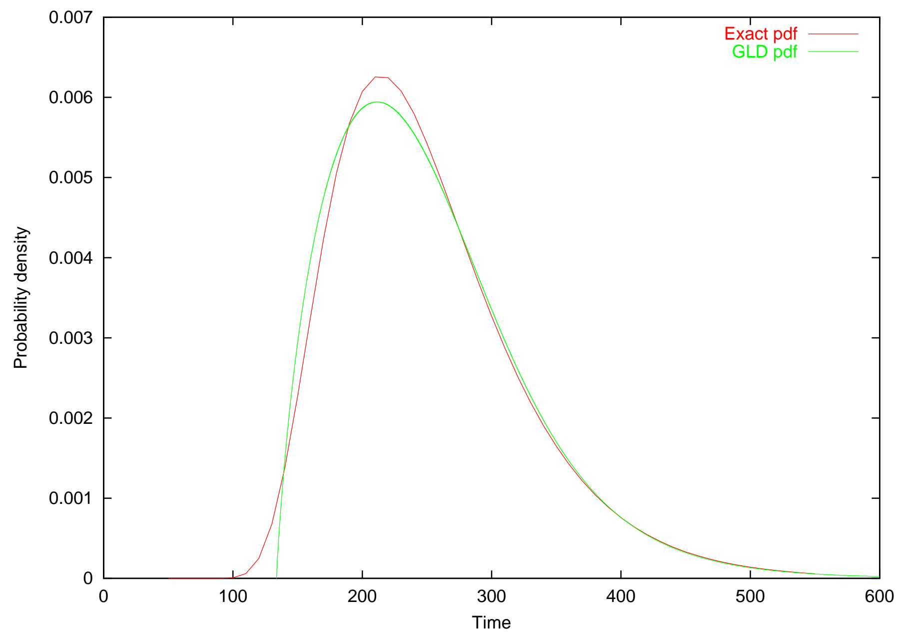

Fig. 4.20. The SMFour model passage time density function produced by the GLD method compared with the exact result.

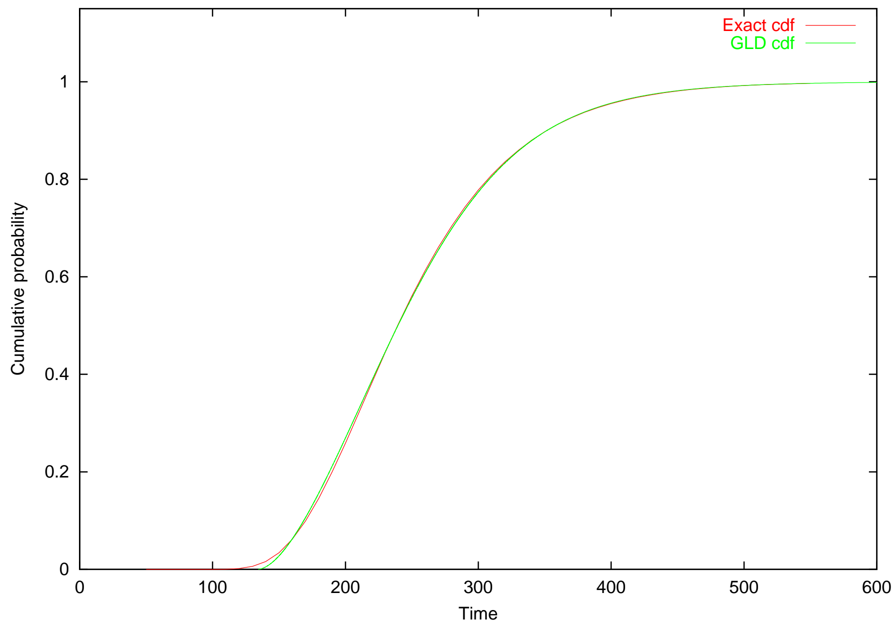

Fig. 4.21. The SMFour model cumulative distribution function produced by the GLD method compared with the exact result.

Chapter 5

Techniques for Analysing Large Models

When performing passage time analysis, the biggest constraint on the size of model which can be analysed is typically the amount of main memory available on the machine in use. For very large models with state spaces of the order of 10^7 states and greater, it is not possible to store the corresponding sparse transition matrix and solution vector in main memory on a single computer as they are too large.

Historically, there have been a number of ways of surmounting this problem, especially in the context of the steady-state solution of large Markov chains:

- Disk-based or out-of-core techniques [46, 89, 93] store the transition matrix and solution vector on disk. The steady-state solution is performed using iterative methods such as Gauss-Seidel, which accesses the elements of the matrix and vector in a sequential fashion. The algorithm proceeds by reading a block of the matrix and a portion of the vector from disk into memory, performing the calculations on this data and then writing the resulting updated portion of the solution vector back to disk. This continues until the algorithm has converged.

  The constraints on the size of models which can be analysed are therefore the disk space available to store them and the amount of time the user is prepared to wait for the solution to be calculated on a single processor. It is usually the

case that disk capacity greatly exceeds memory capacity on modern computers (at the time of writing, most desktop PCs have 512MB of RAM and hard-disks larger than 80GB). However, a significant overhead in disk-based schemes is the time taken to read data from and write it to the disk, which takes far longer than performing the same operations to and from memory. The use of multiple threads or processes to perform computation and I/O concurrently mitigates this somewhat, but high CPU utilisations are rarely achieved.

- Implicit state-space representation techniques [38, 47, 74, 94] seek to reduce the amount of memory required to store the sparse transition matrix by encoding it using symbolic data structures such as Multi-Terminal Binary Decision Diagrams (MTBDDs) [62]. Such symbolic representations can sometimes be stored using less memory than an explicit sparse matrix representation of the same matrix (e.g. one which uses compressed sparse row (CSR) format, see [117] p. 153 for details), especially if the matrix has a regular non-zero structure. This permits the analysis of very large models on relatively modest machines: for example, in [45] a system with 110 million states is represented using only 13.7MB of memory, while in [37] a CTMC with 11 261 376 states requires just 176.1KB of storage using MTBDDs compared with 1 376MB for an explicit scheme.

  However, implicit representation techniques suffer from a number of drawbacks. In particular, operations using an MTBDD representation of a matrix or a vector (such as sparse matrix–vector multiplication) are much slower than the same operations performed using an explicit representation. For example, in the FMS case study available on the PRISM website (see [94]), for the 6 520 state example each iteration of the Jacobi solution method takes 6.45 seconds when using an MTBDD representation and only 0.0024 seconds when using an explicit sparse matrix and vector storage scheme. The difference is observed because of the overheads imposed in the manipulation and maintenance of the implicit representations of the matrix and the vector.

  When using iterative techniques such as Jacobi, Gauss-Seidel or SOR, the values in the iteration vector are updated after each step, which requires the recomputa-

tion of its MTBDD representation. Also, to be efficient MTBDDs require there to be only a few distinct elements within the vector – but this is typically not the case in practice. This imposes an even greater performance overhead which can be avoided by storing the vectors explicitly; but this limits the size of model which can be analysed.

- Parallel and distributed techniques [16, 32, 88] harness the combined memory capacity and computing power of a network of workstations or a dedicated parallel computer to analyse very large models. As the transition matrix is too large to be held within the memory of a single machine, this approach partitions the matrix across a number of processors so that each holds a portion small enough to fit into memory. Calculation of steady-state probabilities and passage time measures in this way requires communication between the processors involved and it is important, therefore, that the partitioning be done in such a way that this communication does not become too large an overhead.

- State-space reduction techniques aim to reduce the size of a model's state-space so that it can be analysed using less memory. This is typically achieved by aggregating states in the underlying stochastic process together and hence reducing the dimensions of the transition matrix. They either operate at the level of the underlying state-transition system directly or they can attempt to aggregate elements in the high-level model (for example, in Petri nets the places and transitions [58]) to reduce the size of the underlying state-space.

It is the two final methods that we consider in more detail in the remainder of this chapter. A number of sparse matrix partitioning strategies are considered, but we favour hypergraph partitioning as it minimises communication whilst maintaining a good balance of computational load. We undertake a comparison between hypergraph partitioning and a simpler technique which only balances computational load and demonstrate that the additional overhead imposed by calculating the hypergraph partition is compensated for by the reduced time taken to perform the matrix–vector multiplications.

This chapter also describes a state-level algorithm which can be applied to semi-Markov models to reduce the dimensions of their state-spaces. We present a practical investigation into the computational aspects of this method, before concluding with a discussion of the suitability of this algorithm for parallel implementation.

5.1 Sparse Matrix Partitioning Strategies

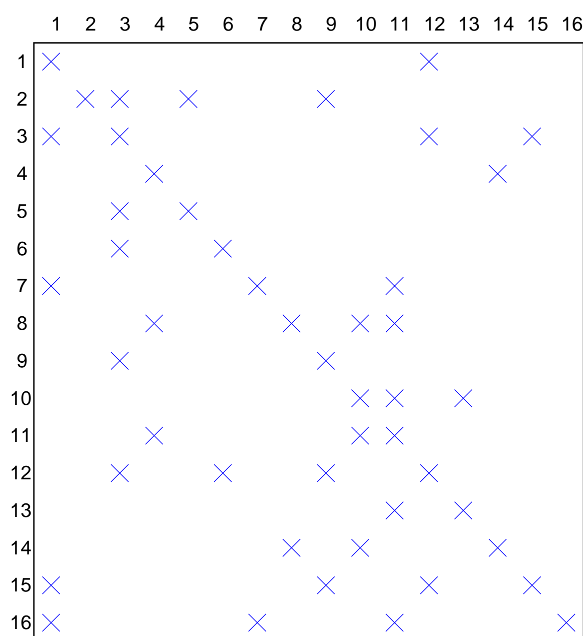

Fig. 5.1. A 16×16 non-symmetric sparse matrix $\mathbf{A}$ [50].

The key opportunity for parallelism in the iterative passage time and transient analysis algorithms presented in Chapter 4, and also in the uniformization computations in Chapter 3, is the sparse matrix–vector multiplications of the general form $\mathbf{Ax}$. We will illustrate the discussion of partitioning schemes which follows with reference to the 16×16 non-symmetric sparse matrix $\mathbf{A}$ illustrated in Fig. 5.1.

To perform sparse matrix–vector multiplications efficiently in parallel it is necessary to map the non-zero elements of $\mathbf{A}$ and $\mathbf{x}$ onto processors such that the computational load is balanced and communication between processors is minimised. As each

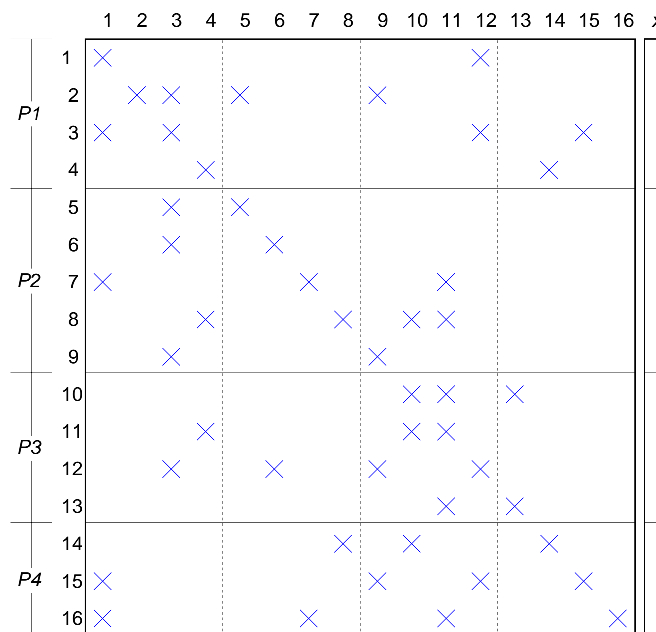

Fig. 5.2. The 4-way row-striped partition of the matrix $\mathbf{A}$ in Fig. 5.1 and the corresponding partition of the vector $\mathbf{x}$.

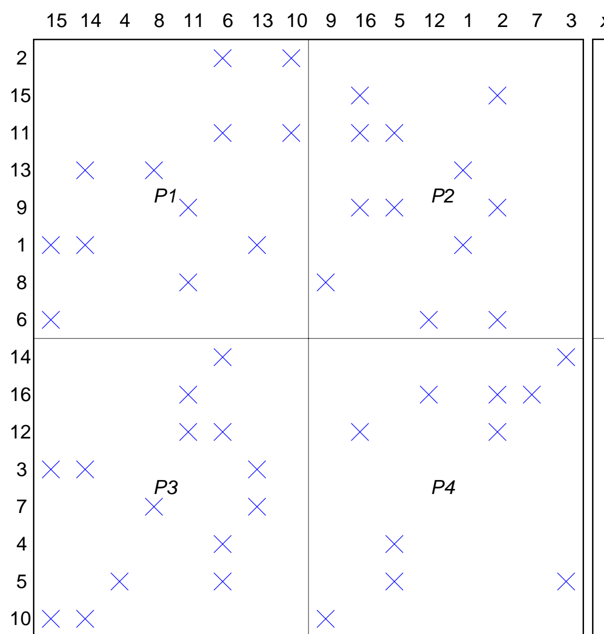

Fig. 5.3. The 4-way 2D checkerboard partition of the matrix $\mathbf{A}$ in Fig. 5.1 (with random asymmetric row and column permutation) and the corresponding partition of the vector $\mathbf{x}$.

processor holds only a portion of the vector which may not contain all the elements that must be multiplied with the locally-stored non-zero matrix elements, this requires processors to communicate newly-computed vector elements to other processors after each iteration. The simplest partitioning strategy is to assign blocks of contiguous rows to the processors such that each processor stores the same number of non-zeros – a *linear* or *row-striped* partition. For p processors and n matrix rows, the first processor is assigned rows $\{1, 2, \ldots, p1max\}$, the second is assigned $\{p1max + 1, \ldots, p2max\}$ and so on. This partition has the advantage of being very easy to compute and also of achieving good load balance, the drawback being that it does not minimise communication.

The effect of applying a row-striped partition to matrix $\mathbf{A}$ of Fig. 5.1 is shown in Fig. 5.2. Note how the number of rows assigned to the processors is not necessarily the same ($P2$ has 5 rows, while $P4$ has 3) but that the number of non-zeros is nearly the same ($P1$ and $P3$ both have 12, $P3$ has 13 and $P4$ has 11). It was not possible to assign the same number of non-zeros to every processor in this case because of the structure of the matrix. The number of off-diagonal non-zeros in this decomposition is 27.

Another option proposed in [108] is to permute the rows and columns of the matrix randomly and then perform a 2D checkerboard partitioning [92]. This approach is inspired by the fact that 2D checkerboard partitioning yields significantly higher efficiencies than row-striped partitioning when applied to dense matrices. For an $n \times n$ sparse matrix partitioned over p processors, this scheme achieves excellent load balance and an asymptotic worst-case communication overhead, per iteration, of $2\sqrt{p}(\sqrt{p} - 1)$ messages of length $n/\sqrt{p}$, giving a total communication volume of $2n(\sqrt{p} - 1)$. The alternative 2D checkerboard algorithm presented in [73] has communication requirements of $2p(\sqrt{p} - 1)$ messages of length n/p, yielding the same total communication volume. The corresponding worst-case communication overhead for a random row-striped partitioning is $p(p - 1)$ messages of length n/p, giving a total communication volume of $n(p - 1)$.

Fig. 5.3 shows the effect of asymmetrically permuting the rows and columns of ma-

trix $\mathbf{A}$ (cf. Fig. 5.1) randomly and applying a 2D checkerboard partition. Note how the structure of $\mathbf{A}$ has been destroyed but that good balance of non-zeros has been achieved (12 each for processors $P1$ and $P2$, 14 for $P3$ and 10 for $P4$). Besides incurring a relatively high per-iteration communication cost, the asymmetric row and column permutations increase the number of operations performed during matrix–vector multiplication. This is because it becomes necessary to reorder vector elements at the end of every iteration.

The disadvantage of all of the above approaches is that they are not scalable because their communication volume exceeds $O(n)$ while, in the context of Markov modelling, the computational cost is usually $O(n)$. This is because, typically, the number of non-zero elements in each row of the matrix (corresponding to the number of transitions out of a state) does not increase significantly with n.

An alternative approach is to apply graph-based partitioning techniques to a row- or column-striped decomposition in order to minimise interprocessor communication whilst maintaining a uniform balance of non-zero elements. In the following, we consider traditional graph-based and recent hypergraph techniques.

5.1.1 Graph Partitioning

In a row-striped decomposition, the $n \times n$ sparse matrix $\mathbf{A}$ can be represented as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each row i $(1 \leq i \leq n)$ in the matrix corresponds to vertex $v_i \in \mathcal{V}$ in the graph. The corresponding weight w_i of vertex v_i is the total number of non-zeros in row i. For the edge-set $\mathcal{E}$, edge e_{ij} connects vertices v_i and v_j with weight $w_{ij} = 1$ if either one of $a_{ij} > 0$ or $a_{ji} > 0$, and with weight $w_{ij} = 2$ if both $a_{ij} > 0$ and $a_{ji} > 0$ [35].

The task of allocating the n rows of matrix $\mathbf{A}$ to p processors is well known to be equivalent to a p-way partitioning of the corresponding graph [82]. The quality of such a decomposition is judged with respect to two metrics: edge cut and balance. An edge is *cut* if the vertices which it connects are assigned to two different processors – so that the total number of edges cut is an approximation for the amount of interprocessor

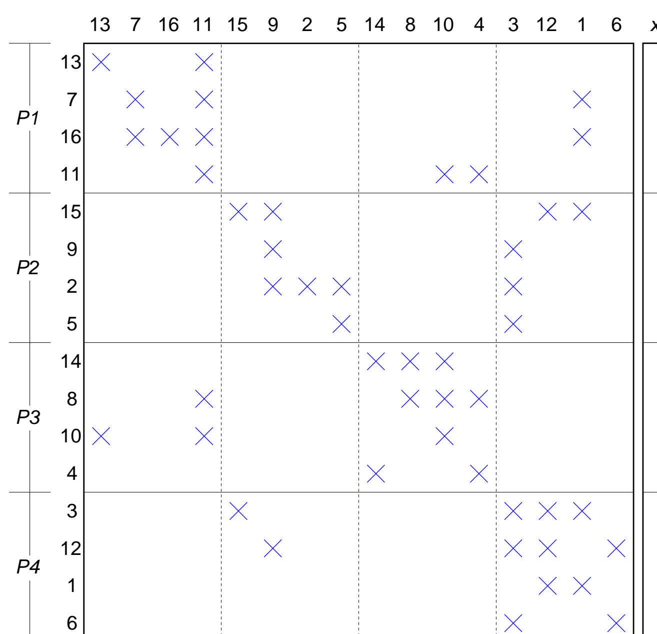

Fig. 5.4. The 4-way graph partition of the matrix **A** in Fig. 5.1 and the corresponding partition of the vector **x** [50].

communication. A decomposition is said to be *balanced* if the sum of the weights of the vertices in each partition does not differ from the average of these weight sums by more than a specified amount. An optimal decomposition is one which minimises edge cut while satisfying the balance constraint.

The problem of finding the optimal decomposition for a given graph is NP-complete. However, there exist a number of tools which implement heuristic algorithms to calculate good sub-optimal decompositions, for example Chaco [72] and MeTiS [80, 82]. A parallel implementation of MeTiS called ParMeTiS [79, 83, 84] is also available. ParMeTiS is particularly attractive for very large matrices as an arbitrary number of processors may be used to calculate the p-way partition, and per-processor memory use is inversely proportional to the number of processors.

Consider the problem of producing a 4-way row-wise decomposition of the matrix shown in Fig. 5.1. The matrix in Fig. 5.4 shows the matrix and vector corresponding to a partition produced by the graph partitioning tool Chaco. Note how the effect of the decomposition has been to minimise the number of non-zeros that occur in off-

diagonal blocks (just 14 off-diagonal elements as opposed to 27 in the original matrix). However, while the edge cut is 14, the number of vector elements that must be sent between processors (i.e. the real communication cost) is just 10. This is because off-diagonal non-zeros which are in the same column and on the same processor are all multiplied by the same remote vector element, a factor which is not accounted for by graph-based partitioning strategies.

5.1.2 Hypergraph Partitioning

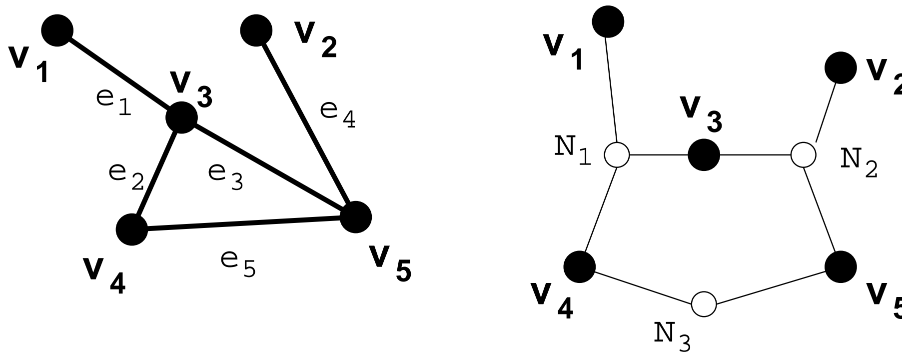

Fig. 5.5. A graph (left) and a hypergraph (right) [121].

Hypergraph partitioning is an extension of graph partitioning. Its primary application has been in VLSI circuit design where the objective is to cluster pins of devices such that interconnect is minimised.

Formally, a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined by a set of vertices $\mathcal{V}$ and a set of nets (or hyperedges) $\mathcal{N}$, where each net is a subset of the vertex set $\mathcal{V}$ [17, 18]. As illustrated in Fig. 5.5, a hypergraph is therefore a generalised graph data structure in which edges can connect arbitrary non-empty subsets of vertices.

In the context of a row-wise decomposition of a sparse matrix as described in [35], matrix row i ($1 \leq i \leq n$) is represented by a vertex $v_i \in \mathcal{V}$ while column j ($1 \leq j \leq n$) is represented by net $N_j \in \mathcal{N}$. The vertices contained within net N_j correspond to the row numbers of the non-zero elements within column j, i.e. $v_i \in N_j$ if and only if $a_{ij} \neq 0$. Weights are assigned to vertices in the same manner as to the vertices of a graph i.e. the weight of vertex i is given by the number of non-zero elements in

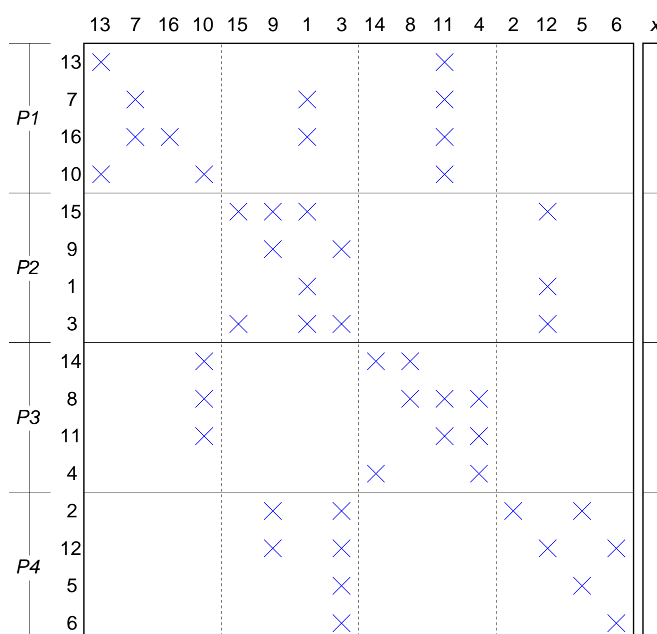

Fig. 5.6. The 4-way hypergraph partition of the matrix $\mathbf{A}$ in Fig. 5.1 and the corresponding partition of the vector $\mathbf{x}$ [50].

row i. The weight of all nets is one, with a net's contribution to the hyperedge cut being defined as one less than the number of different partitions (in the row-wise decomposition) spanned by that net. The overall objective of a hypergraph sparse matrix partitioning is to minimise the hyperedge cut while maintaining a balance criterion. This corresponds to minimising the total communication volume whilst maintaining computational load balance when performing sparse matrix–vector multiplication in parallel. In this context, we apply a hypergraph partition to the corresponding matrix by symmetrically permuting the rows and columns of the matrix such that all rows corresponding to vertices in a partition are assigned to one processor.

The matrix in Fig. 5.6 shows the result of applying hypergraph partitioning to matrix $\mathbf{A}$ of Fig. 5.1. Although the number of off-diagonal non-zeros has increased from 14 to 18 compared with the graph decomposition, the number of vector elements which must be transmitted between processors (the communication cost) has dropped from 10 to 6. This is because hypergraph partitioning algorithms not only aim to concentrate the non-zeros on the diagonals but also strive to line up the off-diagonal non-zeros in

columns. The edge cut of the decomposition is also 6, and so the hyperedge cut exactly quantifies the communication cost, unlike the edge cut in graph partitioning. This is a general property and one of the key advantages of using hypergraphs.

Like graph partitioning, hypergraph partitioning is NP-complete. However, there exist a small number of hypergraph partitioning tools which implement fast heuristic algorithms, for example PaToH [35, 36] and hMeTiS [81, 83]. These are all written to run on a single processor so their capacity is limited to models with a few million states. A current area of research is the development of a scalable parallel hypergraph partitioner – for promising preliminary work in this area see [121, 122, 123]. We note that, for very large models, a parallel graph partitioner still yields a great reduction in communication costs over other methods.

5.1.3 Evaluation

p	Hypergraph partitioning (s)	PC time (s)	Viking time (s)	Row-striped partitioning (s)	PC time (s)	Viking time (s)
1	N/A	2528.3	3993.7	N/A	2528.3	3993.7
2	2.54	1332.3	2358.3	0.47	1495.3	2374.6
4	4.56	780.0	1285.5	0.44	972.0	1239.1
8	6.57	430.8	683.7	0.44	698.1	657.6
16	8.77	323.4	352.8	0.44	553.2	384.5
32	10.95	313.8	190.7	0.45	544.5	219.7

Table 5.1. Run-times for hypergraph partitioned and row-striped parallel sparse matrix–vector multiplication for the analysis of 165 s-points in the 249 760 state Voting model using the iterative algorithm of Chapter 4.

Partitioning a sparse matrix for parallel sparse matrix–vector multiplication using hypergraph partitioning aims to reduce the amount of data which must be exchanged at each step. A key consideration, however, is how much time is saved by doing this – in particular, is it quicker simply to perform a row-striped partitioning and then do the

p	Hypergraph partitioning (s)	PC time (s)	AP time (s)	Row-striped partitioning (s)	PC time (s)	AP time (s)
1	N/A	325.0	1243.3	N/A	325.0	1243.3
2	66.96	258.7	630.5	6.07	635.3	817.4
4	197.12	197.1	328.2	5.61	569.4	484.9
8	266.39	143.0	182.3	5.65	388.3	283.0
16	323.29	114.6	99.7	5.92	362.9	163.0

Table 5.2. Run-times for hypergraph partitioned and row-striped parallel sparse matrix–vector multiplication for the 1 639 440 state FMS model using uniformization (512 multiplications).

p	Hypergraph		Row-striped	
	PC speedup	Viking speedup	PC speedup	Viking speedup
1	1.00	1.00	1.00	1.00
2	1.90	1.69	1.69	1.68
4	3.24	3.11	2.60	3.22
8	6.00	5.84	3.62	6.07
16	7.82	11.32	4.57	10.39
32	8.06	20.94	4.64	18.18

Table 5.3. Speedup figures for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of 165 s-points in the 249 760 state Voting model.

p	Hypergraph		Row-striped	
	PC speedup	AP speedup	PC speedup	AP speedup
1	1.00	1.00	1.00	1.00
2	1.26	1.97	0.51	1.52
4	1.65	3.79	0.57	2.56
8	2.27	6.82	0.84	4.39
16	2.84	12.47	0.90	7.63

Table 5.4. Speedup figures for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of the 1 639 440 state FMS model using uniformization.

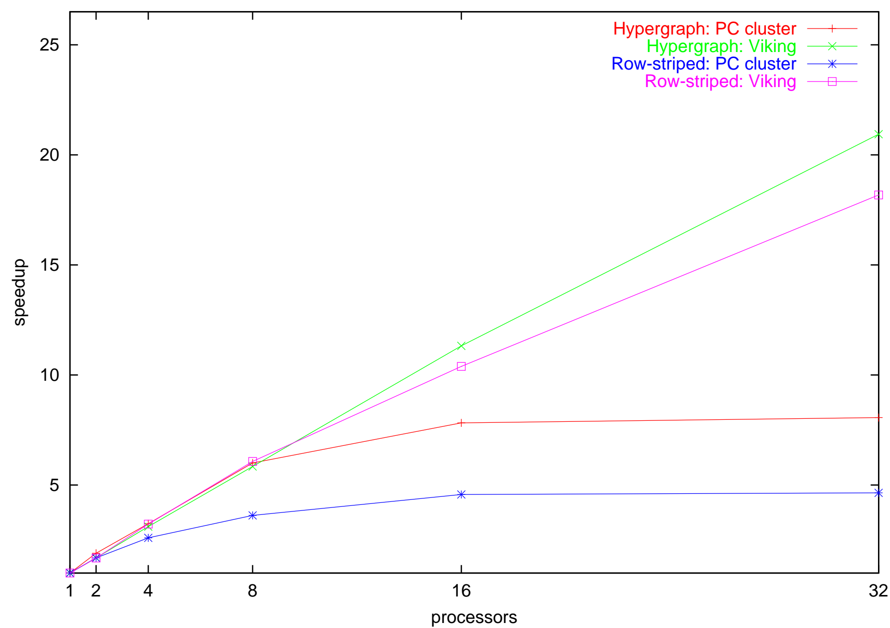

Fig. 5.7. Speedup for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of 165 s-points in the 249 760 state Voting model.

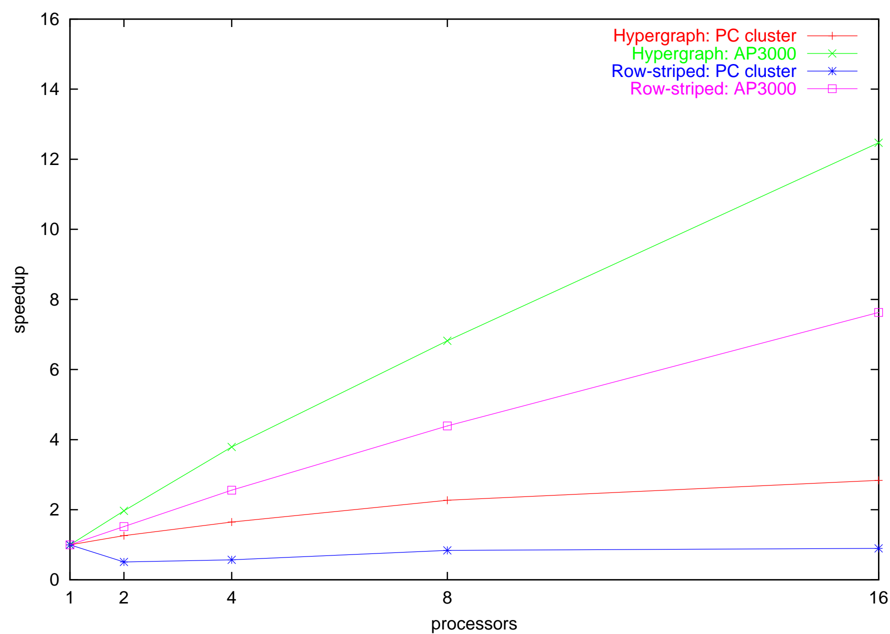

Fig. 5.8. Speedup for hypergraph partitioned and row-striped matrix–vector multiplication for the analysis of the 1 639 440 state FMS model using uniformization.

multiplications at higher cost than it is to calculate a hypergraph partition and then use it in the multiplications?

Table 5.1 compares the partitioning time and multiplication time for hypergraph partitioned and row-striped matrix–vector multiplication on two different architectures for the analysis of 165 s-points derived from the Voting model with $249\,760$ states (see Appendix A.4) using the iterative algorithm of Chapter 4. The PC cluster is a network of workstations, consisting of 32 Intel Pentium 4 2.0GHz PCs each with 512MB RAM linked together by a 100Mbps (megabits per second) switched Ethernet network. The Viking is a Beowulf Linux cluster with 64 dual-processor nodes, where each node has two Intel Xeon 2.0GHz processors and 2GB of RAM. The nodes are connected by a Myrinet network with a peak throughput of 2Gbps. The partitioning was performed on an Intel Pentium 4 2.6GHz machine with 1GB of RAM. In the case of hypergraph partitioning, the PaToH partitioner was used with the following partitioning options:

```
OCM RA=10 MT=12 WI=1 FI=0.05
```

That is, the hypergraph is derived from a sparse matrix and should be partitioned using the Boundary FM refinement algorithm [57] with Krishnamurthy's multilevel gain [91], the Absorption Clustering Using Pins coarsening algorithm [36] and a permitted imbalance between final partitions of 5%.

On the PC cluster, we observe that the time taken to perform hypergraph partitioning and then do the matrix–vector multiplications is lower than the time taken to compute a row-striped partition and then carry out the matrix–vector multiplications for all numbers of processors. As can be seen in Fig. 5.7, the hypergraph-partitioned multiplication scales better than row-partitioned multiplication on this architecture.

On the Viking (where the network is faster) there is much less difference between the speeds of hypergraph partitioned and row-striped multiplication and both methods exhibit similar scalability (although the hypergraph partitioned multiplication does still scale slightly better; see Fig. 5.7). As mentioned in Section 5.1.2, a current area of research is the development of a scalable parallel hypergraph partitioner and so we can expect the overhead of calculating the partition to reduce. At present, the time to cal-

culate the row-striped partition is much lower than the time to calculate the hypergraph partition.

Table 5.2 compares the partitioning time and multiplication time for hypergraph partitioned and row-striped matrix–vector multiplication on a second network of workstations and the Fujitsu AP3000 parallel computer for the analysis of the FMS model with $1\,639\,440$ states (generated with model parameter $k = 7$; see Appendix A.2) using uniformization. The PC cluster is a vanilla network of workstations, consisting of 32 Athlon 1.4GHz PCs each with 512MB RAM linked together by a 100Mbps switched Ethernet network. The AP3000 is based on a grid of 60 processing nodes, each of which has a UltraSPARC 300MHz processor and 256MB RAM. These nodes are interconnected by a 2D wraparound mesh network that uses wormhole routing and that has a peak throughput of 520Mbps.

As with the results from the Voting model, we observe that on both architectures the run-time for hypergraph-partitioned matrix–vector multiplication is lower than that of linear row-striped multiplication for all numbers of processors. In Fig. 5.8 it is observed that hypergraph-partitioned multiplication scales far better than linear row-striped multiplication on the PC cluster. On the AP3000, where the network is faster and the processors slower, the difference is also substantial.

Note, however, that for large numbers of processors (typically, 8 or more) the time to perform the multiplication and the partitioning is higher for the hypergraph scheme than the row-striped scheme. As for the results from the Voting model presented above, it is also the case that we expect the overhead of hypergraph partitioning to fall as new parallel and distributed techniques are developed.

5.2 State-level Aggregation for Semi-Markov Processes

Rather than developing techniques to store and multiply very large sparse matrices across a number of parallel processors, an alternative approach is to reduce the size of the state-space (and hence the dimensions of the corresponding transition matrices) of the model. This has the effect of reducing the time and space requirements of

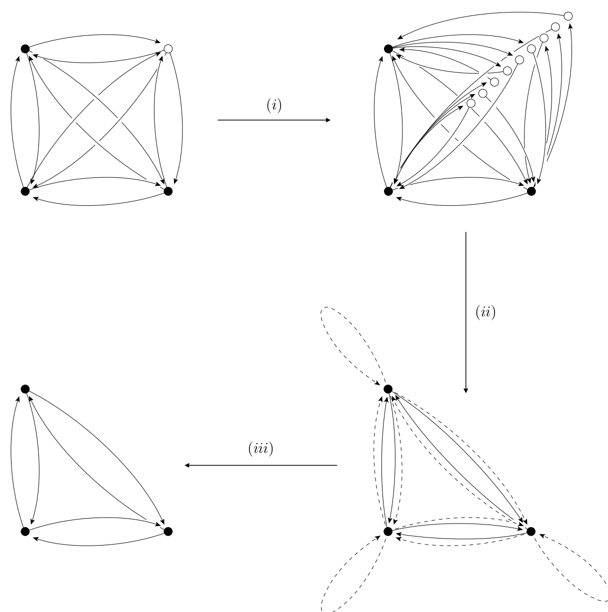

Fig. 5.9. Reducing a complete 4 state graph to a complete 3 state graph.

the passage time analysis to be performed. This is achieved by aggregating states of the model together in such a way that transition probabilities and passage time distributions between unaggregated states are not lost, and therefore the results from the aggregated model are the same as those from the larger, unaggregated model but easier to compute.

We are interested in performing an *exact* aggregation. Many techniques exist in the Markovian domain for exact and approximate aggregation (for example, lumpability [86], aggregation-disaggregation [34], aggregation of hierarchical models [31] and GSPN-level aggregation [58]) but to date analogous work on semi-Markov aggregation algorithms has been very limited. We describe one approach, originally presented in [21], in detail before moving on to consider the cost of performing this aggregation and the effect which it has on the transition matrices with which passage time calculations are performed.

5.2.1 Aggregation Algorithm

The aggregation algorithm of [21] is illustrated in Fig. 5.9. First, a state in the transition graph is chosen to be aggregated. Next, all paths of length two centred on that state are identified (step (i)) and aggregated into stochastically equivalent single transitions (step (ii)). Newly-created transitions (shown dashed in Fig. 5.9) which duplicate the route of existing transitions are combined with the existing transitions. Finally, cyclic transitions are eliminated (step (iii)).

The result is to remove the chosen state and reduce the order of the transition matrix by one. Repeated application of this algorithm will reduce the SMP to an arbitrary size ($\geq$ 2 states) whilst preserving the exact passage time distributions between all pairs of remaining states. Such aggregation is not possible in a Markovian context as aggregation operations of this type do not have a closed form in the Markov domain (i.e. the convolution of two Markovian delays is not itself Markovian).

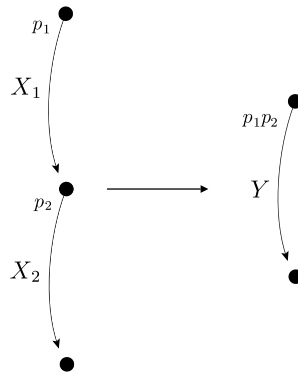

Fig. 5.10. Aggregating sequential transitions in an SMP.

There are three basic reduction steps for aggregating a single state of an SMP. These deal with convolutions, branches and cycles respectively:

Sequential Reduction

In Fig. 5.10, $Y = X_1 + X_2$ is a convolution. In terms of the respective Laplace transforms of the delay functions, this becomes $L_Y(s) = L_{X_1}(s)L_{X_2}(s)$. The probability of the path being selected (delay X_1 followed by delay X_2) is the

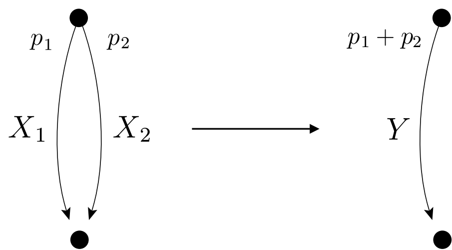

Fig. 5.11. Aggregating branching transitions in an SMP.

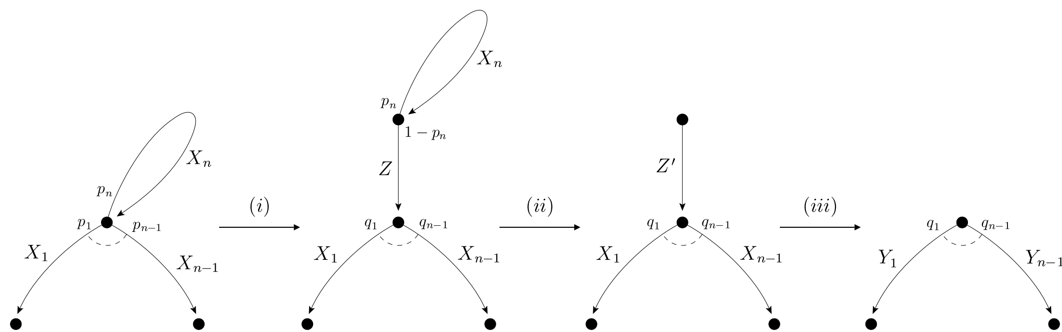

Fig. 5.12. The three-step removal of a cycle from an SMP.

product of the probabilities of the two paths, giving an overall path probability of $p_1 p_2$.

Branch reduction

In Fig. 5.11, the overall selection probability for the aggregated path is the sum of the probabilities of the branches, $p_1 + p_2$. The aggregated distribution Y is given by:

$$L_Y(s) = \frac{p_1}{p_1 + p_2} L_{X_1}(s) + \frac{p_2}{p_1 + p_2} L_{X_2}(s)$$

so that for both aggregated and unaggregated forms the Laplace transform of the total sojourn-time distribution is $p_1 L_{X_1}(s) + p_2 L_{X_2}(s)$. Note that the two branches must have the same start and end state.

Cycle Reduction

When there is a state with at least one out-transition and a transition to itself, as shown in Fig. 5.12, we can remove the cycle by making its stochastic effect part of the out-going transitions. Consider a state transition system which is in the first stage of Fig. 5.12, with $(n-1)$ out-transitions and probability p_i of departure along edge i. Each out-transition has an associated sojourn time distribution function X_i; the cycle probability is p_n with sojourn time distribution function X_n.

The first step (i) is to isolate the cycle and treat it separately from the branching out-transitions. We do this by rewriting the system to include an instantaneous delay and extra state immediately after the cycle, $Z \sim det(0)$; the introduction of an extra state is only to aid our visualisation of the problem and is not performed in the actual aggregation algorithm. Clearly the instantaneous transition will be selected with probability $(1 - p_n)$. We now have to renormalise the p_i probabilities on the branching state to become $q_i = p_i/(1 - p_n)$.

In step (ii) of Fig. 5.12, we aggregate the delay of the cycle into the instantaneous transition creating a new transition with distribution Z'. By treating the system as a random geometric sum of the random variable X_n, we can write:

$$L_{Z'}(s) = \frac{1 - p_n}{1 - p_n L_{X_n}(s)}$$

In stage (iii) of the process, the Z' delay can be sequentially convolved with the X_i sojourn time distributions to give us our final system.

In summary, we have reduced an n-out-transition state where one of the transitions was a cycle to an $(n-1)$-out-transition state with no cycle such that:

$$q_i = \frac{p_i}{1 - p_n}$$

and:

$$L_{Y_i}(z) = \frac{1 - p_n}{1 - p_n L_{X_n}(z)} L_{X_i}(z)$$

Further details of the algorithms used to perform these steps are given in Appendix B. The algorithm removes a single state and reduces the SMP to a normal form, which has no same-state cycles, after each aggregation. If aggregating many states consecutively, an optimisation is to perform this reduction of same-state cycles once only after the last state has been aggregated, rather than after every state aggregation.

5.2.2 State-ordering Strategies

The order in which the states of a semi-Markov system are aggregated can have a significant effect on the space and time demands of the algorithm. In our discussion of various state-ordering strategies, we will focus on two key metrics by which the practicality of these strategies can be judged:

1. the density of non-zero elements in the matrix, also known as the matrix *fill-in*. For an $N \times N$ sparse matrix with r non-zero elements, the density is given by r/N^2.

2. the computational cost of the aggregation process as given by the number sequential, branch and cycle reduction operations performed.

Intuitively, there is a tension between these two metrics. Strategies designed to reduce fill-in should result in more computation as they tend to remove matrix rows with large numbers of non-zeros and so create more transitions which must be combined with

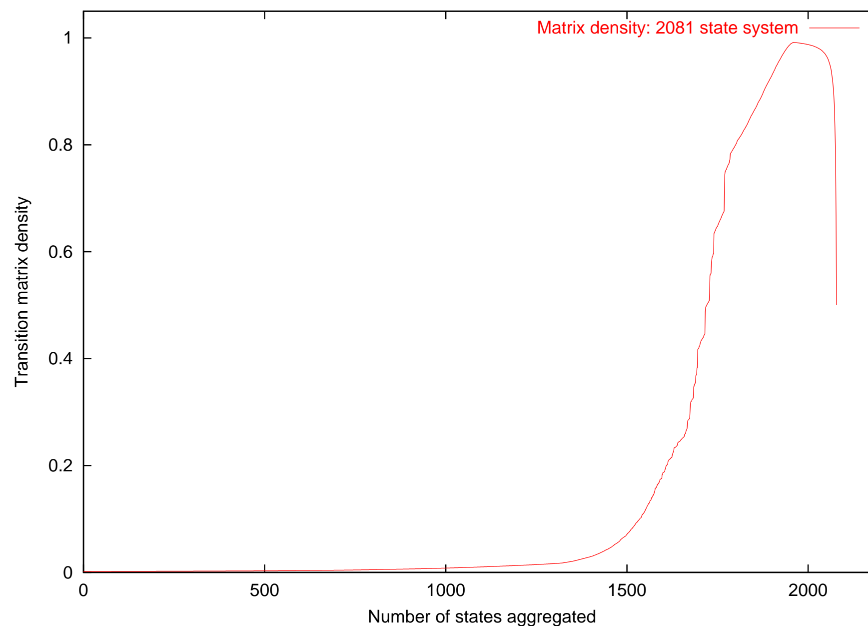

Fig. 5.13. Complete aggregation of a $2\,081$ state semi-Markov system to two states.

those that already exist. On the other hand, strategies designed to reduce computation should tend to increase density faster as they strive to remove states with fewer paths through them. This makes it less likely that the new transitions will coincide with pre-existing ones but will add new non-zeros to the matrix.

Matrix density

Clearly, the density of non-zero elements in the matrix will increase and approach 1.0 as the states are removed and new non-zeros are created in the transition matrix. If a system is aggregated so that only two states remain (with no same-state cycles) then the density will initially approach 1.0 and then decay to 0.5 in the final stages. This decrease in density occurs because the final transition matrix will only have non-zeros in elements $L_{12}(s)$ and $L_{21}(s)$ and zeros in $L_{11}(s)$ and $L_{22}(s)$. This can be observed in Fig. 5.13, which shows the matrix density for a $2\,081$ state semi-Markov process as it is aggregated down to two states.

Computation

In terms of computational cost, for a state with m predecessor states and n successor states, there are mn convolution operations and as many as mn branching aggregations to perform. For an $N \times N$ transition matrix with high density, this will give us $O(N^3)$ operations to perform to aggregate $O(N)$ states. For a sparse transition matrix with low density, m and n may be $O(1)$ rather than $O(N)$ and if the newly created transitions do not coincide with existing transitions then this gives us a lower bound of mn convolutions to perform.

As semi-Markov models are often generated from high-level formalisms such as Petri nets and process algebras where the number of potential successors of a state is almost always limited, their initial matrices tend to be sparse. As aggregation proceeds, however, these matrices will become increasingly dense, with the computational cost consequences for the ultimate passage time solution that this entails. In order to gain any benefit from aggregation, therefore, it may be necessary to curtail the process before the improvements to solution time gained from reducing the state space size are outweighed by the cost of performing the aggregation.

5.2.3 State-selection Algorithms

Bearing in mind the transition matrix density and computational complexity issues highlighted above in Section 5.2.2, we propose the following orderings of states for aggregation. Given that a state has m predecessor states and n successor states:

Fewest-paths-first chooses the state with lowest mn-value first. This is designed to minimise computation, as $O(mn)$ convolution and branching aggregations are required to eliminate a state.

Most-successors-first chooses the state with the highest n-value first. This is designed to reduce fill-in by targeting the rows of the transition matrix with the largest number of non-zeros for aggregation.

Most-paths-first chooses the state with highest mn-value first. This is designed to demonstrate the computationally worst-case scenario.

Random chooses an arbitrary state for aggregation, without consideration of m or n. The selection is done uniformly across the entire state space, giving us a yardstick with which to compare our other state-ordering strategies.

5.2.4 Comparing Aggregation Strategies

To demonstrate the aggregation technique we use the example semi-Markov model of a Voting system (described in Appendix A.4). We generate four different sized state-spaces from this example: $2\,081$ and $4\,050$ state models which are used to test several different aggregation strategies, and $106\,540$ and $541\,280$ state models which are aggregated using the algorithm identified as most efficient.

Fig. 5.14 shows the change in density of the transition matrix for fewest-paths-first, most-successors-first, most-paths-first and random state-selection methods when aggregating all but two states in the $2\,081$ state model. Note that, as with all remaining graphs in this section, the results are plotted with a logarithmic y-axis. It can be seen that the fewest-paths-first method provides the lowest density for nearly 75% of the aggregation process. The most-paths-first technique maintains a lower transition matrix density for the last 25% of the process.

The most-successors-first strategy experiences a density explosion early on, reaching 70% fill-in relatively quickly, almost certainly because it generates a large number of new non-zero elements at the start of the process. This is clearly demonstrated by its density profile when compared to that of the random state-selection method for the first half of the state-space. Once 35% of states have been aggregated, however, the desired effect is achieved as the removal of dense rows starts to lower the fill-in.

Fig. 5.15 shows the cumulative number of convolution, branch elimination and cycle elimination operations taken to aggregate all but two states in the $2\,081$ state model. The fewest-paths-first policy maintains a very low operations count for the first 70% of the state space. The number of operations performed by the most-paths-first algorithm

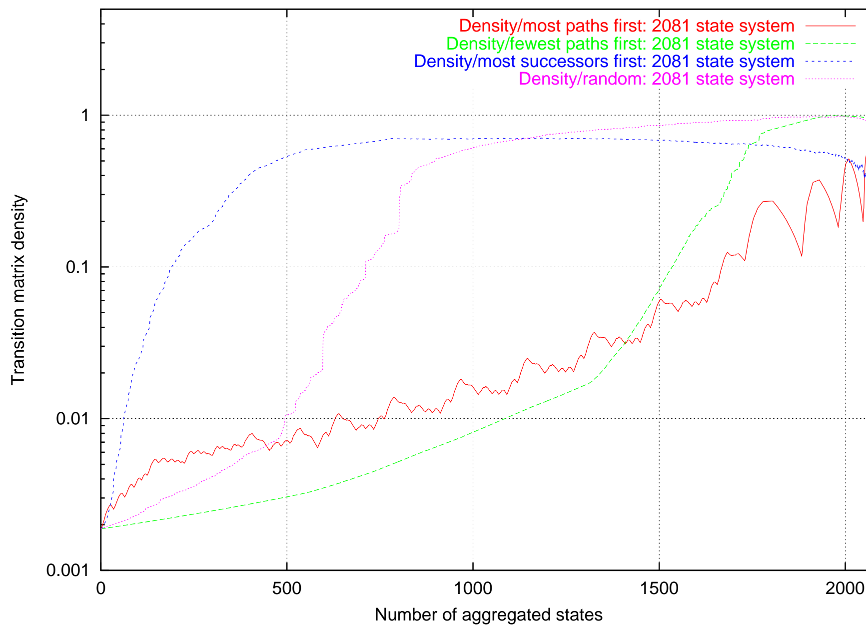

Fig. 5.14. Transition matrix density for the 2 081 state model for four different state-selection algorithms.

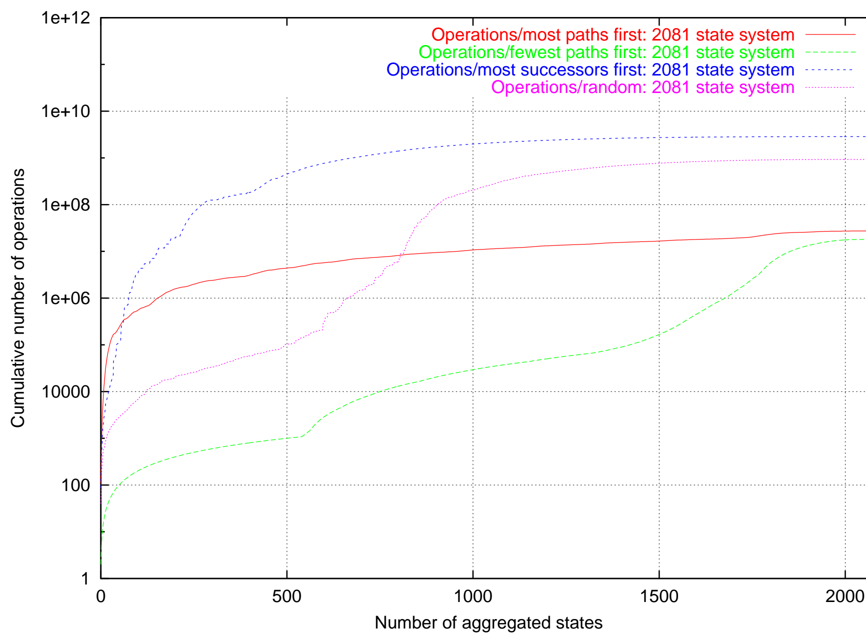

Fig. 5.15. Computational cost (in terms of sequential, branching and cycle reduction operations) for the 2 081 state model for four different state-selection algorithms.

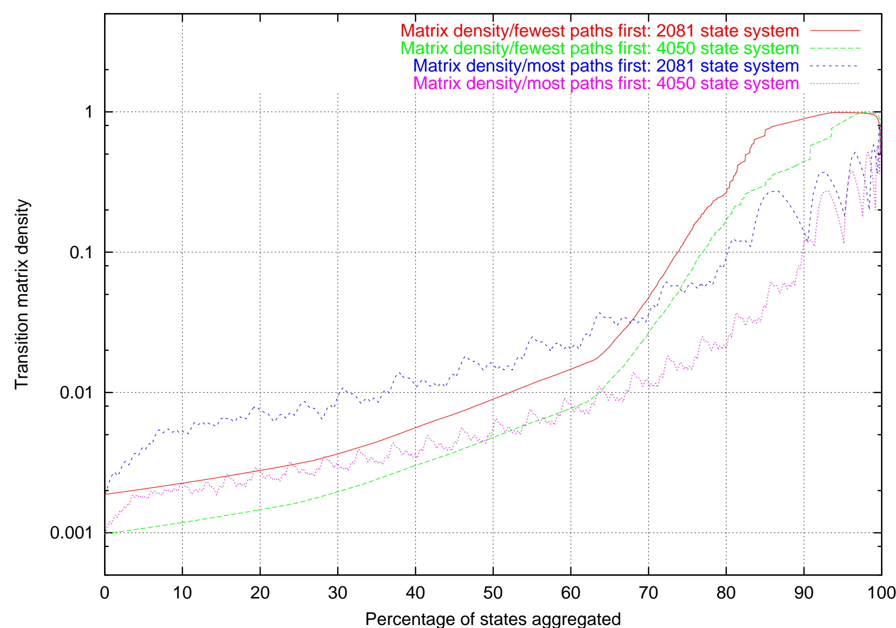

Fig. 5.16. Transition matrix density over two different model sizes and two different state-selection algorithms.

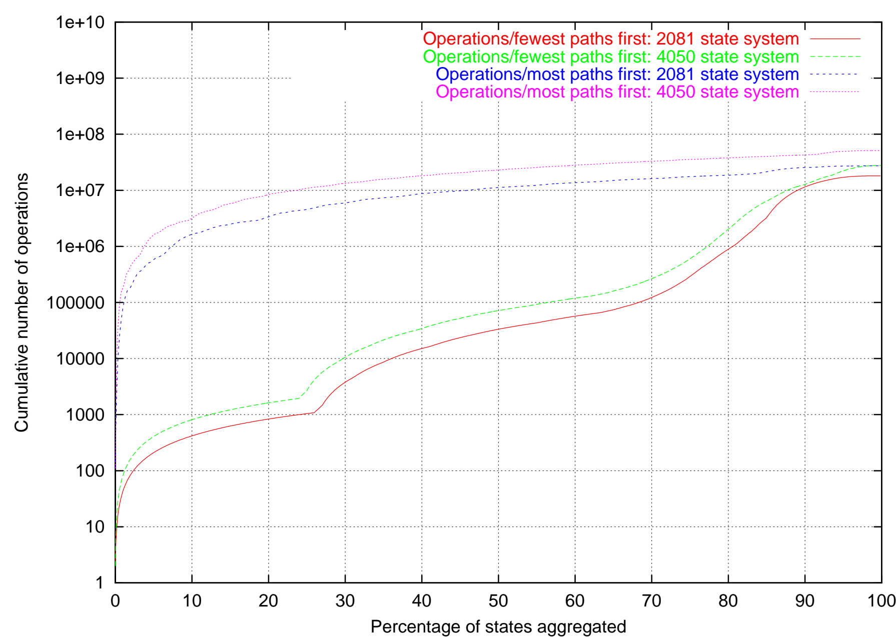

Fig. 5.17. Computational complexity over two different model sizes and two different state-selection algorithms.

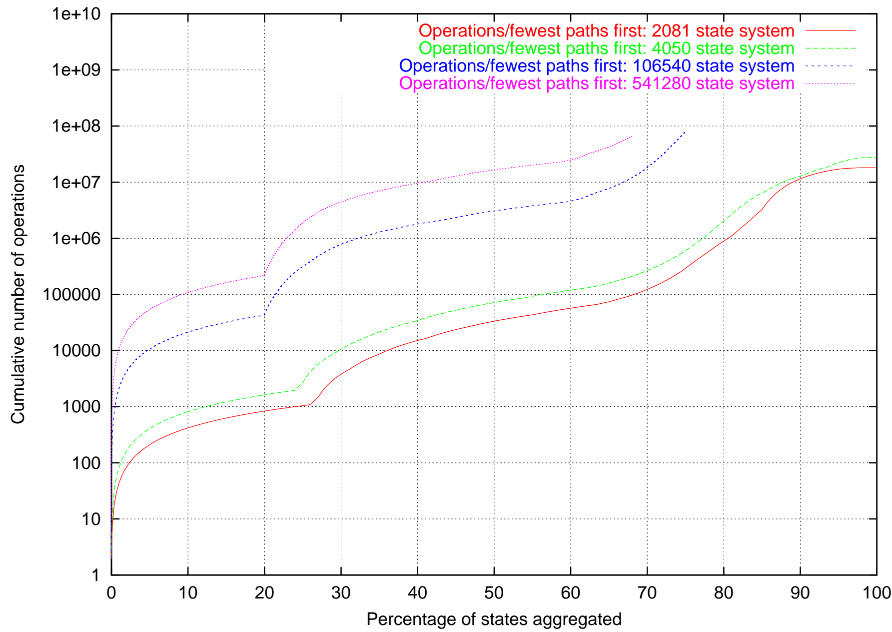

Fig. 5.18. Computational complexity for systems with up to 541 280 states; fewest-paths-first algorithm only.

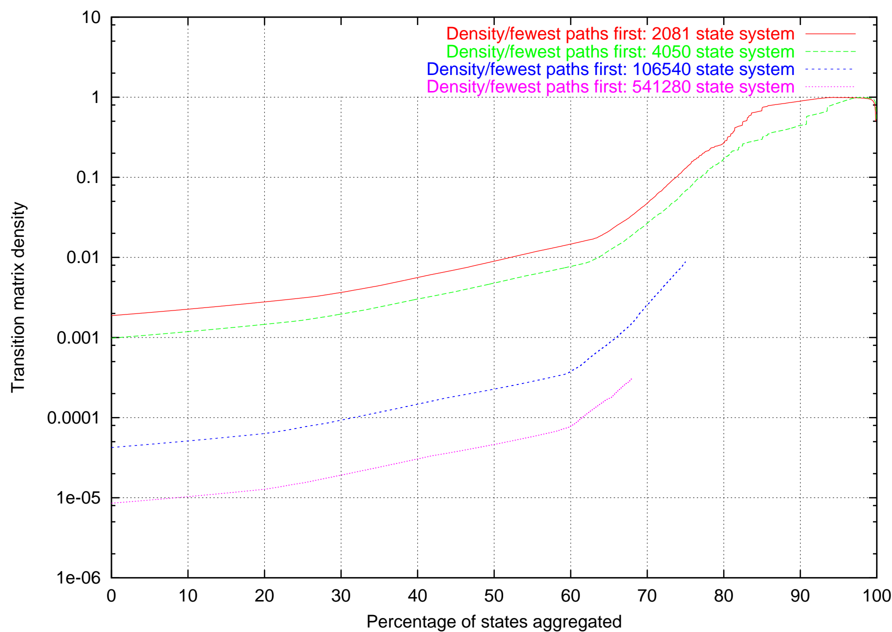

Fig. 5.19. Transition matrix density for systems with up to 541 280 states; fewest-paths-first algorithm only.

appears linear in the number of states aggregated. Ultimately, the most-paths-first policy performs twice as many operations as the fewest-paths-first algorithm. The most-successors-first policy is also seen to be computationally worse (that is, result in a higher number of operations) than the random policy. For this reason it will not be considered further for the aggregation of larger state spaces.

5.2.5 Comparing Models of Different Size

In Fig. 5.16, we compare density profiles over two different sizes of model ($2\,081$ and $4\,050$ states) for two different aggregation strategies: fewest- and most-paths-first algorithms. For a valid comparison to be made, we plot the density against percentage of state space aggregated. The results show that, after about 75% of states have been aggregated, the transition matrix density is lower for the larger model, right up until complete matrix fill-in is achieved.

The computational cost for different model sizes is shown in Fig. 5.17 for fewest- and most-paths-first aggregation techniques. For both techniques there is a small increase in the cost when aggregating larger models; but this is dwarfed by the orders-of-magnitude increase that can be seen when using the most-paths-first algorithm over the fewest-paths algorithm.

In Fig. 5.18 and Fig. 5.19 we consider only the fewest-paths-first algorithm, as, of all the techniques, it seems to be the one which keeps the computational cost under control the longest whilst simultaneously maintaining matrix sparsity. In addition to the two previous cases, we provide results from the aggregation of SMPs with $106\,540$ and $541\,280$ states.

Fig. 5.18 shows the number of operations for all four state spaces. Our previous results suggest that our first concern should be over the amount of computation to be done – whilst matrix sparsity remains at acceptable levels when a large proportion of the state space has been aggregated, the number of operations exceeds 10^6 even when small state spaces are aggregated completely. For larger models, therefore, the process is truncated after about 75% of the state space has been aggregated, avoiding the com-

putational explosion. Some success with this truncation can be observed as it limits the required number of aggregation operations for both $106\,540$ and $541\,280$ state systems to below 10^8 (this is of the same order as the number of operations required to aggregate the $4\,050$ state model completely using the most-paths-first method).

Finally, Fig. 5.19 shows the effect on matrix density for models with up to $541\,280$ states, stopping when 75% of the state space has been aggregated. Again the density of the larger model remains smaller for longer and even for the $106\,540$ state model only reaches 0.01. Sparse-matrix solution techniques will still function well at such densities, and will benefit greatly from the reduction in the dimensions of the matrix.

5.2.6 Parallel Aggregation

One possible drawback of the aggregation algorithm above is the amount of computational effort required to aggregate the majority of the state-space of a large model. It could be the case that the time taken to perform the aggregation and then analyse the aggregated model is greater than the time required to analyse the original unaggregated model. This stems from the fact that an exact aggregation which considers only one state at a time is performed. In order to reduce the amount of time taken, therefore, it is necessary to investigate parallel approaches to aggregation.

The general scheme would be to take the very large state-space, partition it across a number of processors and have each processor perform aggregation of the portion which they have been allocated before recombining the aggregated pieces of the state-space and proceeding with the passage time analysis on a single machine. In this case, the fact that the algorithm only works on a single state at a time becomes a major benefit as it means that different processors can work on different portions of the state-space without affecting each other. This is because the aggregation of a single state affects only the immediate predecessors and successors of the chosen state. States whose immediate predecessors or successors belong to other processors may be excluded from aggregation, however, as this would require the communication of a large amount of information between processors. In order to be able to aggregate any significant num-

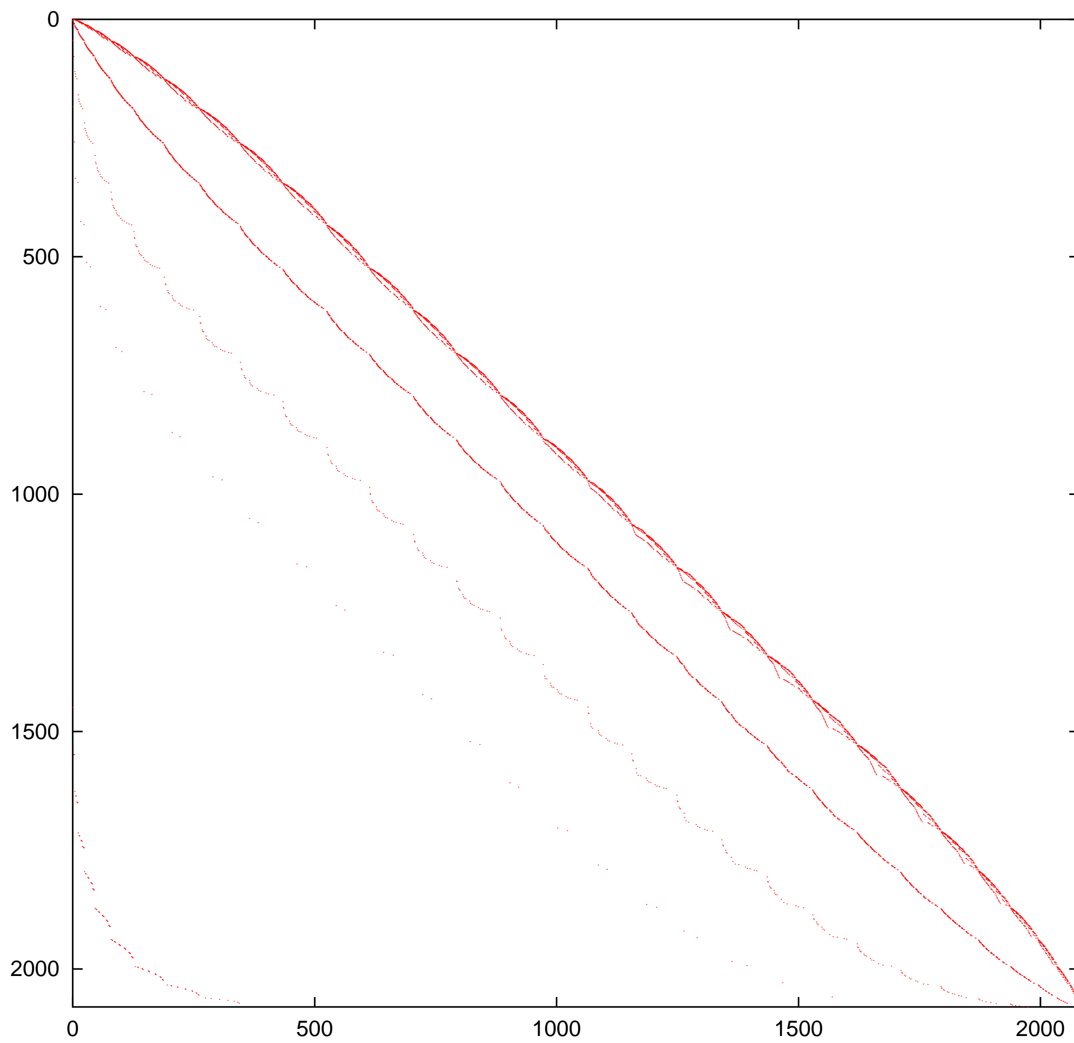

Fig. 5.20. Transition matrix for the 2 081 state Voting model.

Fig. 5.21. Hypergraph bi-partitioned transition matrix for the 2 081 state Voting model.

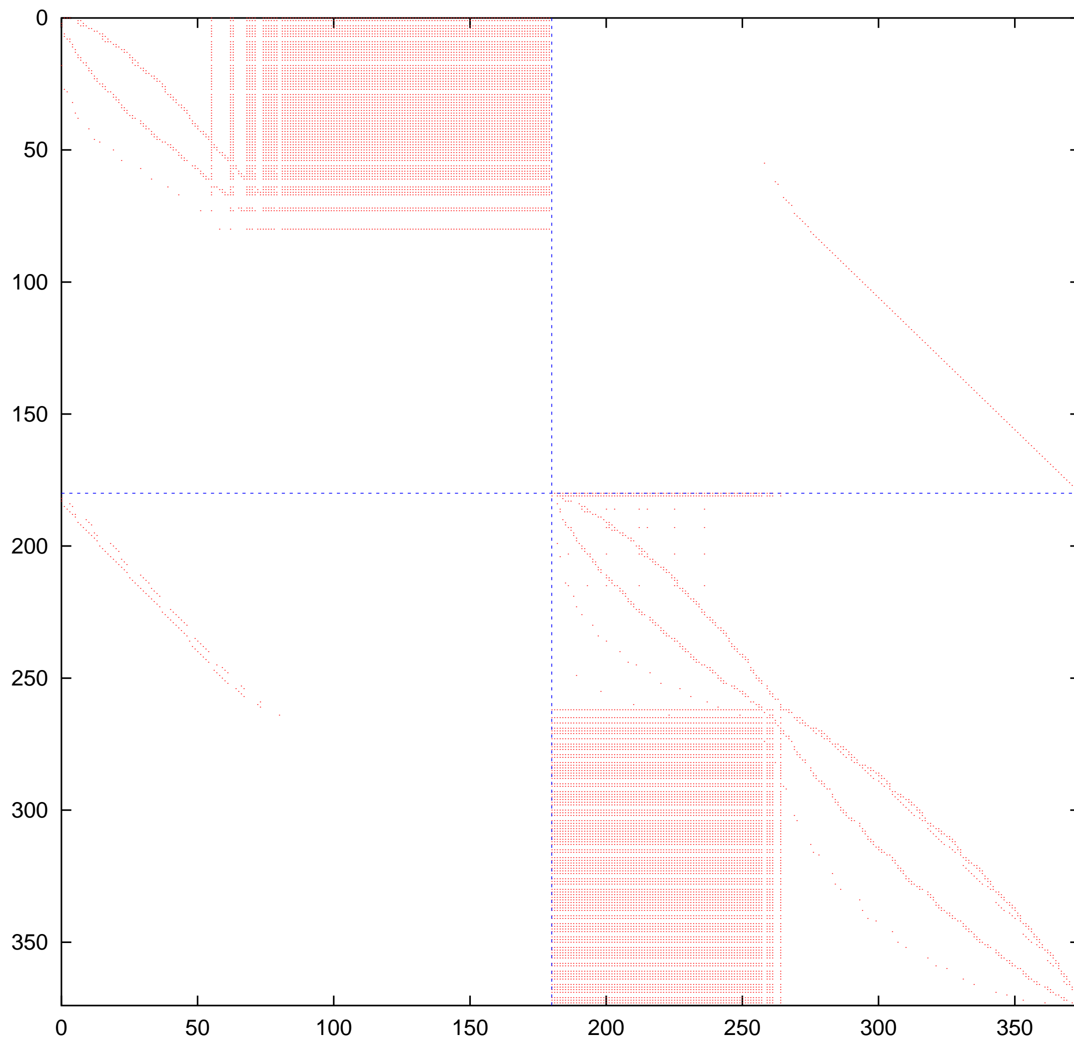

Fig. 5.22. Aggregated hypergraph-partitioned transition matrix for the $2\,081$ state Voting model (contains 374 states).

ber of states, we must minimise the number states to which this condition applies as much as possible.

We can determine which states are not eligible for aggregation by examining the model's transition matrix. Assuming the matrix is partitioned by allocating entire rows to different processors, a state has a predecessor belonging to a different processor if there is a non-zero in the corresponding column in a row not allocated to the processor which is responsible for that state. Similarly, a state has a successor which belongs to a different processor if there is a non-zero in its row of the matrix which corresponds to a row-number not belonging to the current processor. In order to minimise the number of states which cannot be aggregated we are required to minimise the number of off-diagonal non-zeros in the partitioned matrix. This can be achieved by employing hypergraph partitioning.

The effect of hypergraph partitioning on the layout of the $2\,081$ state Voting model's transition matrix can be seen in Figs. 5.20, 5.21 and 5.22. Initially, the breadth-first state-generation algorithm (cf. Section 6.1.5) creates the transition matrix as seen in Fig. 5.20 with a distinctive almost-lower-diagonal layout. This is then partitioned using a hypergraph partitioner, resulting in the layout shown in Fig. 5.21 for a two-way partition. The aggregation is then conducted on the two partitions independently, excluding states which have either immediate predecessors or immediate successors in the other partition. The resulting reduced transition matrix with 374 states is shown in Fig. 5.22. Note how the density pattern has changed dramatically, with the on-diagonal blocks for both processors becoming more dense.

One potential drawback with this scheme is that as the number of partitions increases, the number of states eligible for aggregation will decrease. Table 5.5 shows the percentage of the state-space which can be aggregated without affecting states in other partitions for a number of different models and number of partitions. It can be seen that the percentage of eligible states does decrease for an increasing number of partitions, but that this effect is more pronounced for small models than for large ones. This is beneficial as, because of the dimension of their transition matrices, it is precisely these large models which we would wish to reduce in size the most and which

Model	States	Number of processors				
		2	4	8	16	32
Voting-1	2 081	82.3%	67.7%	40.0%	27.7%	18.7%
Voting-2	4 050	89.9%	83.2%	68.6%	41.8%	26.8%
Voting-3	106 540	96.2%	93.6%	88.1%	78.7%	69.9%
Voting-4	249 760	97.7%	96.0%	92.8%	86.2%	79.4%
Voting-5	541 280	98.1%	96.8%	94.2%	88.9%	83.7%
Voting-6	778 850	98.4%	97.2%	95.0%	90.4%	84.3%
Voting-7	1 140 050	98.6%	97.6%	95.7%	91.7%	86.0%
Web-server-1	107 289	98.0%	90.8%	85.1%	76.0%	69.0%
Web-server-2	248 585	93.2%	91.6%	82.3%	77.8%	73.4%
Web-server-3	517 453	98.2%	92.1%	83.3%	79.9%	76.2%
Web-server-4	763 680	93.9%	89.6%	83.8%	81.4%	78.6%
Web-server-5	1 044 540	94.4%	89.4%	86.1%	81.9%	78.7%

Table 5.5. Percentage of state-space which can be aggregated without requiring information stored on remote processors. Shown for differing numbers of partitions in a number of different sized models.

would need to be divided across as many processors as possible.

We conclude by noting a disturbing trend which requires future investigation: despite the impressive reduction in the dimensions of the matrix, the number of non-zeros in the aggregated matrix of Fig. 5.22 ($15\,726$) is actually larger than the number in the original matrix of Fig. 5.20 ($8\,120$). This has serious implications for the amount of memory which will be consumed in storing the matrix, and also the amount of time it will take to perform calculations using it. This suggests that when performing aggregation, it is important to keep track of the number of non-zeros in the matrix after each new state removal. If it becomes higher than the original matrix, then only states whose removal would not create any more new non-zeros should be aggregated. We identify this as a future area of research, particularly when seeking to perform aggregation efficiently in parallel.

Chapter 6

Implementations

This chapter describes the implementation of three tools based on the algorithms and techniques described in Chapters 3, 4 and 5 for the calculation of steady-state probabilities, passage time densities and transient distributions in very large Markov and semi-Markov models.

We begin by describing the Markovian DNAmaca steady-state solver [87] on which these implementations are based. We then describe HYDRA (HYpergraph-based Distributed Response-time Analyser) [49] which enables the calculation of passage time densities and transient distributions in Markov models through the use of uniformization. Hypergraph partitioning is employed to permit the efficient parallel analysis of very large state spaces.

Next, we detail SMCA (Semi-Markov Chain Analyser), a semi-Markov extension of DNAmaca for the steady-state analysis of large semi-Markov chains. To the best of our knowledge, this is the only such analyser to have been written for the semi-Markov domain. This required an extension of DNAmaca's input language to support the specification of generally-distributed transition firing delays, and corresponding alterations to the state generator and steady-state solver. These modifications are described in this chapter.

Finally, we present SMARTA (Semi-MArkov Response Time Analyser), a parallel analysis pipeline which implements the iterative algorithms of Chapter 4 for the anal-

ysis of passage time and transient measures in very large semi-Markov models. This builds upon SMCA by making use of the augmented model description language, state-space generator and steady-state solver. A distributed Laplace transform inverter implementing the Euler method is employed to numerically invert the Laplace transform of the required performance measure. The central computation to be performed in this step is repeated sparse matrix–vector multiplications, and we use hypergraph partitioning to implement this efficiently in parallel.

6.1 DNAmaca

DNAmaca is a Markov chain steady-state analyser with the proven ability to analyse models with states spaces larger than $O(10^7)$ states [87]. We describe it here as it forms the basis of the three tools implemented as part of our work. DNAmaca supports models specified as SPNs, GSPNs, SPAs and queueing networks.

The architecture of DNAmaca is shown in Fig. 6.1. A model description, which describes the structure of the model and the steady-state performance measures of interest, is parsed into C++ code and the result is compiled with a pre-existing probabilistic hash-based state-space generator library. Running the resulting executable generates the state-space and generator matrix $\mathbf{Q}$ of the model's underlying CTMC on disk with vanishing states eliminated. This state-space is checked by a functional analyser to test if it is irreducible. If it is not, then it may be possible to eliminate transient states and remap the remaining states so that it becomes irreducible. The steady-state distribution of the CTMC is then calculated using one of a number of solution methods (specified in the input file). Finally, the C++ code generated from the model description is linked with a pre-compiled performance analyser library and executed to compute the steady-state performance measures requested by the user. We consider each of the steps shown in Fig. 6.1 in more detail in the sections which follow.

Model
Description

Parser

User
Code

State Space
Generator

State Space

Performance
Analyser

Results

Transition
Matrix

Common Library
Routines

Steady State
Solution

Functional
Analyser

Steady State
Solver

Fig. 6.1. DNAmaca tool architecture [87].

6.1.1 Model Specification Language

DNAmaca's model specification language has a $\TeX$-like syntax and makes use of standard C/C++ expressions such as comparisons and variable assignments. The language defines a high-level model in terms of its resources and the customer population at each resource. Transitions, which are enabled or not according to the current populations of certain resources, can fire and in doing so alter the populations on the resources in the model. This is obviously a natural way to describe stochastic Petri nets and queueing networks, but it is also flexible enough to permit the specification of stochastic process algebra models [6, 22, 23]. The user specifies the performance measures of interest using the same language.

It is not intended that this section should provide an exhaustive description of DNAmaca's input language; instead we focus on only the central portion needed to describe most models. Interested readers are directed to [87, 88] for the full details of the language.

In the description which follows we use the following symbols:

$\{X\}*$	denotes one or more occurrences of X
$\vert$	separates alternatives

6.1.2 Model Description

A model description consists of:

```
model_description = \model{
  {
    state_vector | initial_state | transition_declaration |
    constant | solution_control
  }*
}
```

State Descriptor Vector

States are described by an array of values which are modified by the firing of transitions. When describing Petri nets, this array maps naturally onto the array of places in the net – the value of each entry in the state descriptor vector tracks the number of tokens on the corresponding place. The state descriptor vector is composed of one or more C/C++ variables, usually of type `int`, `long`, `short` or `char`. The names of these variables must conform to C/C++ naming rules for variables. That is to say, each name must be a sequence of one or more letters, digits or underline symbols which does not begin with a digit or clash with any pre-existing keyword.

```
state_vector= \statevector{
  { <type> <identifier> {, <identifier> }*; }*
}

type = C/C++ variable type
identifier = C/C++ variable name
```

Initial State

It is necessary to supply a description of the initial state of the model to provide the state-space generation process with a starting point. This is done by assigning values to the elements of the state vector in the same way as assigning values to variables in C/C++; e.g. having already declared a variable `int a` in the state descriptor vector, the value of 1 is assigned to it by `a = 1`.

```
initial_state = \initialstate{
  { <assignment> }*
}

assignment = C/C++ assignment statement
```

Transition Declarations

The dynamic behaviour of the model is captured by the enabling and firing of its transitions. For each transition, it is necessary to specify the following:

- the conditions under which the transition will become enabled. This is a boolean expression in terms of some or all elements of the state descriptor vector.
- the change in the state of the model which results from the transition firing. This specifies the changes to the values in the state descriptor vector which result when the transition fires.
- either the probabilistic weight of the transition, for immediate transitions, or the exponentially-distributed firing rate, for timed transitions.
- the (optional) priority level of the transition, which permits higher-priority transitions to pre-empt the firing of those with lower priority.

A transition description in the input language is composed of the following:

```
transition_declaration = \transition{<identifier>}{
  \condition{ <boolean expression> }
  \action{ { <assignment> }* }
  \rate{<real expression>} | \weight{<real expression>}
  \priority{ <non-negative integer> }
}
```

Constants

Constant values which are used repeatedly throughout a model description (for example the number of customers in a closed queueing network) can be defined.

```
constant = \constant{ <identifier> }{value}
```

6.1.3 Solution Control

These options allow the user to specify the method and parameters to be used when performing steady-state analysis (including for the analysis undertaken when calculating the relative weights of the source states for passage time analysis). A full list of supported methods is given in Section 6.1.7 below.

```
solution_control = \solution{
  {
    \method{ <method name> } | \accuracy{ <real> } |
    \maxiterations{ <long int> }
  }*
}

method name = { gauss | grassman | gauss_seidel | sor |
                bicg | cgnr | bicgstab | bicgstab2 |
                cgs | tfqmr | ai | air | automatic }
```

6.1.4 Performance Measure Specification

Two types of steady-state performance measure are supported in DNAmaca. These are *state measures* and *count measures*:

```
steady_state_measures =  \performance{
  { state_measure | count_measure }*
}
```

State measures are expressed in terms of the values of the elements of the state vector and can be used to compute measures such as the average number of tokens on a place in a Petri net (corresponding perhaps to the average number of jobs in a buffer in the system being modelled) or the probability of being in a subset of the state-space. We can compute the mean, the variance, the standard deviation and the probability distribution of such measures.

```
state_measure = \statemeasure{identifier}{
  \estimator{ {mean | variance | stddev | distribution}* }
  \expression{ <real expression> }
}
```

Count measures allow event frequencies such as the throughput of transitions to be computed. The user can specify pre- and post-conditions on the event which must hold for the measure to be evaluated and must provide a list of transitions for which the measure will be computed.

```
count_measure = \countmeasure{identifier}{
  \estimator{mean}
  \precondition{ <boolean expression> }
  \postcondtion{ <boolean expression> }
  \transition{ all | { <identifier> }* }
}
```

6.1.5 State-space Generator

The parsing of the high-level model description and performance measure specification produces C/C++ code for a `State` class. Elements of the model description then become attributes and member functions of this class; for example, the state vector elements become attributes of `State` of the type declared in the state descriptor vector and there is a function `fire()` which alters the values of these attributes exactly as specified in the transition declarations. This code is then separately compiled and linked with a general-purpose state-space generator library. The resulting executable explores the underlying state-space of the high-level model, generating the transition matrix **Q** and a list of the state descriptor vectors for all states in the state-space.

A breadth-first search is employed to explore the model's state-space using the algorithm described in [88] and outlined here in Fig. 6.2. This starts from an initial state s_0 (specified in the model in the `\initialstate` clause) and uses a FIFO queue

```
begin
    A = ∅
    E = s_0
    F.insert(s_0)
    while F(not empty) do begin
        F.remove(s)
        foreach s' ∈ succ(s) do begin
            if s' ∉ E do begin
                F.insert(s')
                E = E ∪ s'
            end
            A = A ∪ id(s) → id(s')
        end
    end
end
```

Fig. 6.2. Breadth-first search algorithm for state-space exploration [88].

(F) and a list of explored states (E) to generate the state graph (A). The functions `insert()` and `remove()` add a state to and extract a state from F respectively, while `succ(s)` returns the set of successor states of s. The function `id(s)` returns a unique sequence number for state s. Breadth-first search is favoured over depth-first search as it allows **Q** to be generated and written to disk row-by-row without the need to maintain more than one row in memory.

When generating very large state-spaces a major problem is keeping track of which states have already been explored – it is important not only to keep the amount of memory consumed by E low but also to be able to search E quickly to determine if a newly generated state has been encountered before. The most naïve method of implementing E would be to maintain it as an array or linked list of full state vectors. This has the advantage of being *exhaustive* as every encountered state is guaranteed to be in the list and so a newly generated state will not be erroneously treated as having already been explored. There are two obvious limitations, however. Firstly, to maintain a list of all previously encountered state vectors can require a large amount of memory, especially when there are many states and/or the individual state descriptor vectors are very large (for example in a Petri net with many places). Secondly, searching the list to check to see if a newly generated state is already there (which must be done repeatedly in the state-space generation process) is time-consuming – for a list of n states it would take at most n state descriptor vector comparisons to ascertain that a new state had not already been encountered.

We use a probabilistic hash-based scheme [87, 88, 90] to overcome both of these problems. Rather than maintain a list of full state vectors, we store hashed values of the state vectors in the rows of a hash table. This requires two independent hash functions: one to calculate in which row the state belongs (the *primary* hash value) and the second to calculate the hashed value of the state vector to be inserted in that row (the *secondary* hash value). Memory usage is reduced over the exhaustive scheme as only the hashed values are stored (the full state descriptor is written to disk and not referred to again during generation), and the time complexity of checking for a state vector hash in the hash table is $O(1)$ versus $O(n)$ for checking for its existence in an n element list.

It is possible, however, for two different states to map onto the same entry in the hash table – when this happens a *collision* is said to have occurred. If this is the case (which can only be if both hashed values are the same) then the second state to be encountered will be incorrectly omitted from the generated state-space as it will have been deemed already to have been entered in the hash table. It is possible to calculate the probability that an incorrect omission will occur in terms of the total number of states, the number of rows in the hash table and the number of distinct values the secondary hash value can take (cf. [87, 88, 90]), and by careful choice of these values ensure that this probability is acceptably low. For example, with $10\,000\,000$ states, $350\,003$ hash table rows and 40-bit hash keys (giving 2^{40} distinct values), the probability of a collision is 0.0001. It is important to note, however, that an omission probability of 0.01% *does not* mean that 0.01% of all states generated will be incorrectly identified as having already been explored and so not feature in the final state-space; rather it means that there is a 0.01% probability that one or more states will be missing from the generated state-space. 99.99% of the time the full correct state-space is generated.

The state-space exploration process results in two files being written to disk, one containing the rows of $\mathbf{Q}$ and the other a list of the state descriptor vectors of all the states in the state-space.

6.1.6 Functional Analyser

Steady-state analysis of a model requires that its state-space be irreducible; that is, that every state be reachable from every other (cf. Definition 2.3). For an arbitrary high-level model there is no guarantee that this will be the case, and so DNAmaca incorporates a functional analyser. This examines the generated state-space using a strongly-connected component checking algorithm [12] and, if the state-space is not irreducible, attempts to map it onto one which is. There are three possible results of this procedure:

- The state-space is irreducible and analysis proceeds to the steady-state solution stage.

- The state-space consists of a number of transient states and a single irreducible component. As these transient states will have a steady-state probability of zero they can be eliminated from the state-space. Steady-state analysis then proceeds on the single irreducible component.
- The state-space consists of several separate irreducible components. This case cannot be remapped onto a single irreducible state-space and so the steady-state solution is not attempted.

6.1.7 Steady-state Solver

The steady-state solution of the model's CTMC is achieved by solving $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$ for $\boldsymbol{\pi}$ (subject to $\sum_i \pi_i = 1$). This can be rewritten in the form of a system of linear equations $\mathbf{Ax} = \mathbf{b}$, for which there exist a wide range of solution techniques. DNAmaca implements a number of techniques which can be grouped into four main categories:

Direct methods: Gaussian Elimination and Grassman's method.

Classical iterative methods: Gauss-Seidel, fixed SOR and dynamic SOR.

Krylov subspace techniques: Conjugate Gradient Squared (CGS), Conjugate Gradient using the normal equations, Biconjugate Gradient, BiCG stabilised (two versions) and the Transpose Free Quasi-Minimal Residual Algorithm.

Decompositional techniques: Aggregation-Isolation and Aggregation-Isolation Relaxed.

Interested readers are directed to [87, 88] for full details of these algorithms and their implementations.

6.1.8 Sparse Matrix Representation

DNAmaca's great strength is its ability to analyse very large state-spaces. Central to this ability are the data structures employed for representing the very large $\mathbf{Q}$ matrices in memory. The tools presented later in this chapter all employ a similar storage

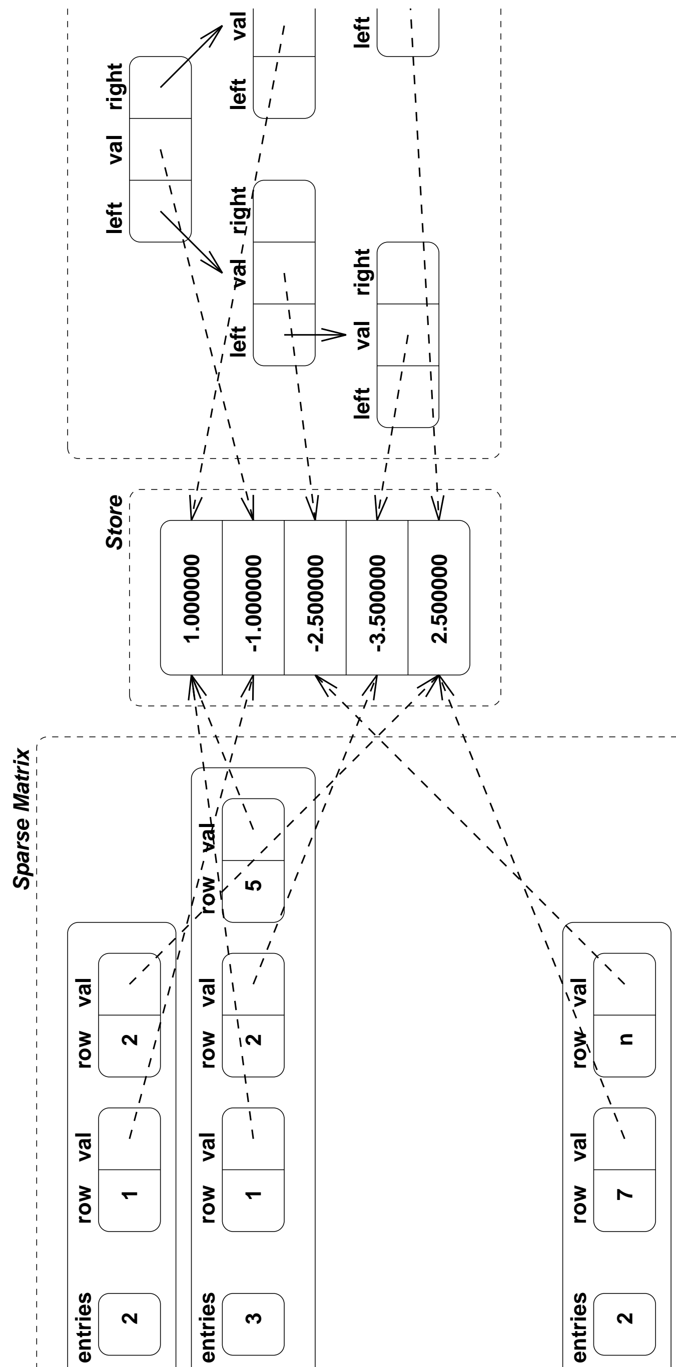

Fig. 6.3. DNAmaca's sparse matrix representation scheme [87].

scheme for their matrix representations and so we describe it in some detail here. The choice of representation is governed by the access pattern of the algorithms to the non-zeros in the matrix and the number of distinct non-zeros stored in the matrix.

Most of the steady-state algorithms and all of the passage time/transient algorithms only require access to their matrices ($\mathbf{Q}$, $\mathbf{P}'$ and $\mathbf{U}$) by row or by column exclusively. This means that the representation can be tailored to give access to either the rows or to the columns of the matrix but does not have to cater for both row and column access. We can therefore store the matrix as a vector of vectors, with the vector at position i storing the non-zeros of row i or column i (depending on the algorithm) of the matrix.

Furthermore, as the models analysed are derived from high-level descriptions with (usually) a small number of transitions, the number of distinct non-zero values in the matrix should be reasonably low – although it is worth noting that this may not be the case when transitions have state-dependent rates or weights. This means that it is wasteful, in general, to store every non-zero explicitly. Rather we can maintain a list of all the distinct values and have a pointer into that list from every non-zero value in the matrix. In the case of matrices $\mathbf{Q}$ and $\mathbf{P}'$ these non-zeros will be floating-point numbers, but in $\mathbf{U}$ they will be complex numbers. In either case, however, the same scheme can be used.

These considerations lead to the creation of the sparse matrix representation structure shown in Fig. 6.3 (reproduced from [87]). There are three components:

Sparse matrix. As described above, this is implemented as a vector of vectors which can be dynamically resized.

Store. This holds the distinct values of the non-zeros in the matrix. Non-zero elements in the sparse matrix have pointers to the corresponding value in this store.

AVL tree. Used only in the construction of the matrix and deleted when no longer needed, the AVL tree is a height-balanced binary tree. It is used to check efficiently if a non-zero value to be added to the matrix already exists in the store. Normally, checking such a list of n elements would require $O(n)$ time, but an AVL tree can be searched in $O(\log_2 n)$ time.

6.1.9 Performance Analyser

DNAmaca can compute two classes of steady-state performance measures as described above in the language specification: state measures and count measures. State measures are expressed in terms of the values of the elements of the state vector and can be used to compute measures such as the average number of jobs in a buffer. Given the steady-state probability distribution vector $\boldsymbol{\pi}$ and a vector of expression values $\mathbf{v}$ (where v_i is a function of the elements in the state descriptor vector of state i), the mean of a state measure m is given by:

$$E[m] = \sum_{i}^{n} \pi_i v_i$$

and its second (raw) moment as:

$$E[m^2] = \sum_{i}^{n} \pi_i v_i^2$$

The variance of m is therefore:

$$Var[m] = \sum_{i}^{n} \pi_i v_i^2 - \left(\sum_{i}^{n} \pi_i v_i \right)^2$$

Count measures are used to calculate the mean rate at which events occur in the model, allowing the computation of quantities such as the throughput of transitions. Given the steady-state probability distribution vector $\boldsymbol{\pi}$ and a function r_i which returns the rate at which the event occurs in state i, the mean of a count measure m is:

$$E[m] = \sum_{i=1}^{n} \pi_i r_i$$

6.2 HYDRA

In this section we describe the first of the tools produced as a result of the work presented in this thesis. HYDRA builds on the technology used by DNAmaca to permit the analysis of very large Markov models for passage time and transient measures through the use of uniformization (cf. Chapter 3). The capacity of DNAmaca to handle very large state-space is expanded further by the use of a parallel approach and

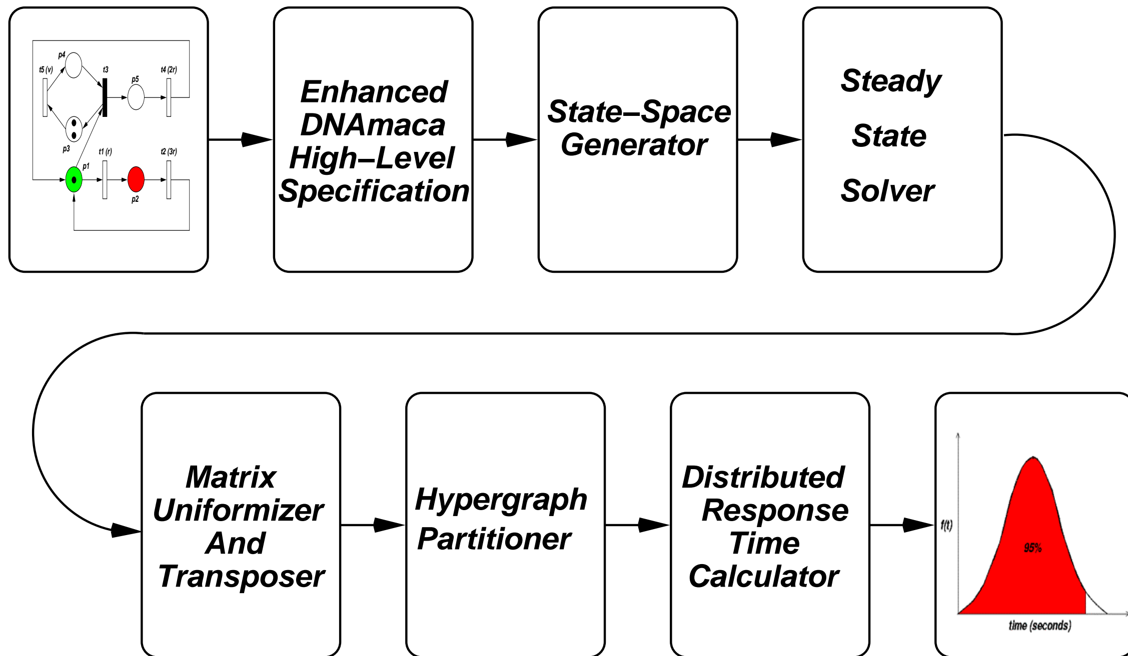

Fig. 6.4. HYDRA tool architecture.

we exploit hypergraph partitioning (cf. Chapter 5) to perform the necessary sparse matrix–vector multiplications efficiently across a number of processors. Results presented in Chapter 8 demonstrate that HYDRA has the ability to analyse models with more than 10^7 states.

Fig. 6.4 shows the architecture of the HYDRA tool. The process of calculating a response time density begins with a high-level model, which we specify in an enhanced form of the DNAmaca Markov chain analyser interface language. Next, the probabilistic, hash-based state generator uses the high-level model description to produce the generator matrix $\mathbf{Q}$ of the model's underlying Markov chain as well as a list of the initial and target states. Normalised weights for the initial states are then determined from Eq. 3.10. We calculate the uniformized matrix $\mathbf{P}$ from $\mathbf{Q}$ (cf. Section 3.3) and construct $\mathbf{P}'^T$ from $\mathbf{P}$ by transposing and making the target states absorbing. Having been converted into an appropriate input format, $\mathbf{P}'^T$ is then partitioned using a hypergraph or graph-based partitioning tool. The analysis pipeline is completed by our distributed passage time and transient calculator.

6.2.1 Input Language Augmentation

The input language used by HYDRA is an extended version of that employed by DNAmaca. As DNAmaca is purely a steady-state solver we have added syntax to permit the specification of passage time and transient performance measures. For either case, the user is required to specify a high-level description of the measures of interest.

For passage time analysis, the user must specify the conditions which identify the source and target states of the passage and also the time range for which the result should be calculated. The time range is specified as an initial value of t, an incremental step and a maximum value. The source and target conditions are expressed as C/C++ boolean expressions in terms of the elements of the state vector of the model.

```
passage_time_measure = \passage{
  \sourcecondition{ <boolean expression> }
  \targetcondition{ <boolean expression> }
  \t_start{ <real expression> }
  \t_stop{ <real expression> }
  \t_step{ <real expression> }
}
```

Conditions for transient measures are expressed in a similar fashion:

```
transient_state_measure = \transient{
  \sourcecondition{ <boolean expression> }
  \targetcondition{ <boolean expression> }
  \t_start{ <real expression> }
  \t_stop{ <real expression> }
  \t_step{ <real expression> }
}
```

6.2.2 State Generator and Steady-state Solver

HYDRA employs the same probabilistic hash-based state-space generator as DNAmaca to generate $\mathbf{Q}$. When there are multiple source states in the passage time measure it is necessary to calculate the steady-state of the CTMC's embedded Markov chain (EMC) by solving $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$ subject to $\sum \pi_i = 1$. This may be rewritten as $\mathbf{A}\mathbf{x} = \mathbf{0}$ where $\mathbf{A} = (\mathbf{I} - \mathbf{P})^T$, $\mathbf{x} = \boldsymbol{\pi}^T$ (where the superscript T denotes the transpose operation) and $\mathbf{0}$ is a vector of zeros, which can then be solved using the techniques outlined in Section 2.1.4 or any of the methods supported by DNAmaca described above in Section 6.1.7.

For very large models it may be necessary to perform these calculations in parallel, which requires the partitioning of $\mathbf{A}$ and $\mathbf{x}$ as described in Chapter 5. When using iterative steady-state solution techniques in parallel, such partitioning requires the exchange of elements of $\mathbf{x}$ at the end of each iteration. This enforced barrier synchronisation limits execution speed to that of the slowest processor.

Asynchronous iterative algorithms attempt to overcome this problem [19, 61]. Rather than exchange vector elements at the end of every iteration, processors send updates to each other every n iterations. These values are then used until the next update is received, meaning that some elements of $\mathbf{x}$ will be out of date compared with those computed locally. This may delay the convergence of the solution, but as it is usually the case that the computation capacity of a network of processors exceeds that network's communication capacity it may be advantageous to increase the amount of computation performed while reducing communication. The use of graph partitioning may be of some benefit when using asynchronous iterations as the amount of non-zeros which must be communicated between processors will be minimised and so the amount of information which must be received from other processors will be low [51].

When calculating passage time measures, the end result of this step is the creation of a file containing $\boldsymbol{\alpha}$, the source-state weighting vector (cf. Eq. 3.10).

6.2.3 Matrix Uniformization and Transposition

Once $\mathbf{Q}$ has been generated and the steady-state solution for the EMC has been calculated, we uniformize (cf. Eq. 3.7) and transpose $\mathbf{Q}$ in preparation for partitioning and passage time analysis. We determine the value of $q > \max_i |q_{ii}|$ and then read in the rows of $\mathbf{Q}$, dividing every non-zero by q and add 1 to the diagonal elements, in order to construct the columns of $\mathbf{P}^T$. At the same time, we ensure that the target states are made absorbing. This gives us our final uniformized and transposed matrix $\mathbf{P}'^T$.

6.2.4 Hypergraph Partitioner

It is necessary to partition the state-space so that the final passage time calculations can be performed in parallel. Writing a hypergraph partitioner was beyond the scope of the work presented here. Instead, any one of a number of off-the-shelf tools such as PaToH and hMeTiS can be used to perform this step (cf. Chapter 5). The output from all the tools is a text file which assigns each row (or column, if doing a column-wise partition) to one of the partitions. This mapping is then applied to $\mathbf{P}'$ and $\boldsymbol{\alpha}$.

6.2.5 Uniformization-based Passage Time and Transient Analyser

The analysis pipeline is completed by our distributed passage time density calculator, which is implemented in C++ using the Message Passing Interface (MPI) [67] standard. This means that it is portable to a wide variety of parallel computers and workstation clusters. Initially, each processor tabulates the Erlang terms for each t-point required (cf. Eq. 3.8). Computation of these terms terminates when they fall below a specified threshold value. In fact, this is safe to use as a truncation condition for the entire passage time density expression because the Erlang term is multiplied by a summation which is a probability. The terminating condition also determines the maximum number of hops m used to calculate the inner summation in Eq. 3.8, which is independent of t.

Each processor reads in the rows of the matrix $\mathbf{P}'^T$ that correspond to its allocated

partition into two types of sparse matrix data structure and also computes the corresponding elements of the vector $\boldsymbol{\pi}^{(0)}$. *Local* non-zero elements (i.e. those elements in the diagonal matrix blocks that will be multiplied with vector elements stored locally) are stored in the sparse matrix representation described above. *Remote* non-zero elements (i.e. those elements in off-diagonal matrix blocks that must be multiplied with vector elements received from other processors) are stored in an ultrasparse matrix data structure – one for each remote processor – using a coordinate format. That is to say, each non-zero is stored in the form `<rowIndex> <columnIndex> <nonZeroValue>`. Each processor then determines the vector elements which will need to be received from and sent to every other processor on each iteration, adjusting the column indices in the ultrasparse matrices so that they index into a vector of received elements. This ensures that a minimum amount of communication takes place and makes multiplication of off-diagonal blocks with received vector elements very efficient.

The vector $\boldsymbol{\pi}^{(n)}$ is then calculated for $n = 1, 2, 3, \ldots, m$ by repeated sparse matrix-vector multiplications of form $\boldsymbol{\pi}^{(n+1)T} = \mathbf{P}'^T \boldsymbol{\pi}^{(n)T}$ (where m is the upper limit on the number of Erlang terms in Eq. 3.8). Actually, fewer than m multiplications may take place since a test for convergence is made after every iteration (cf. Eq. 3.11); if the convergence criterion is satisfied, the matrix-vector multiplication is not performed and we set $\boldsymbol{\pi}^{(n+1)T} = \boldsymbol{\pi}^{(n)T}$ in subsequent iterations. The check for convergence is performed on each processor individually and the results broadcast to every other processor. Only if the calculations on all processors have converged do we stop performing the multiplications. The broadcasting of convergence results is, therefore, a synchronisation point in the algorithm.

For each matrix-vector multiplication, each processor begins by using non-blocking communication primitives to send and receive remote vector elements, while calculating the product of local matrix elements with locally stored vector elements. The use of non-blocking operations allows computation and communication to proceed concurrently on parallel machines where dedicated network hardware supports this effectively. The processor then waits for the completion of non-blocking operations

(if they have not already completed) before multiplying received remote vector elements with the relevant ultrasparse matrices and adding their contributions to the local matrix-vector product cumulatively.

From the resulting local matrix-vector products each processor calculates and stores its contribution to the sum $\sum_{k\in\vec{j}} \pi_k^{(n)}$. After m iterations have completed, these sums are accumulated onto an arbitrary master processor where they are multiplied with the tabulated Erlang terms for each t-point required for the passage time density. The resulting points are written to a disk file and are displayed using the GNUplot graph plotting utility.

6.3 SMCA

The second tool produced as a result of the work described in this thesis is an extension of DNAmaca to semi-Markov processes. Our motivation was that, while there exist a large number analysis tools for Markov models (see for example [39, 94, 114, 116]), relatively little work has been done on the steady-state analysis of large semi-Markov models. We have sought to rectify this by producing SMCA, which is capable of analysing SMPs of the same size state-spaces as the CTMCs solvable with DNAmaca. As with HYDRA, SMCA builds on a large amount of the technology incorporated into DNAmaca such as hash-based probabilistic state-space exploration and efficient sparse matrix representation. We therefore discuss below the modifications necessary to DNAmaca to permit the analysis of SMPs. Much of this feeds into the final tool presented in this chapter, namely the parallel passage time and transient analyser SMARTA.

6.3.1 Input Language Augmentation

The greatest difference when moving from the Markov domain into the semi-Markov domain is that state holding-times are no longer constrained to being exponentially distributed. This required altering DNAmaca's input language to permit the specification

of such transitions, modifying the state-space generator to generate SMPs and incorporating the different approach for steady-state solution of SMPs into the steady-state solver.

The modified DNAmaca input language is tailored towards the description of Semi-Markov Stochastic Petri Nets (cf. Section 2.2.4) although it can also be used to specify any semi-Markov chain. Transitions are described in the same manner as in DNAmaca (cf. Section 6.1.2) except that it is also necessary to describe the firing-time density function associated with each transition. As our analysis is conducted in the Laplace domain, we describe this density function in terms of its Laplace transform. Also, because the inversion algorithms require the values of these Laplace transforms at many values of s, the user is required to provide a C/C++ function which returns the value of the firing-time density function's Laplace transform at a given value of s. Note that the `\rate{}` clause is not used in SMCA as transitions only have probabilistic weights.

To aid the user, macros are defined for several common firing-time distributions as shown in Table 6.1. Note that the `markov()` distribution is a special case; it permits the semi-Markov analyser to model true Markovian concurrency and, if used, there can be no non-Markovian transitions enabled at the same priority level. The weight of a `markov()` transition is used as the rate parameter of its exponential firing-time distribution.

A transition description in the input language is composed of the following:

```
transition_declaration = \transition{<identifier>}{
  \condition{ <boolean expression> }
  \action{ { <assignment> }* }
  \weight{ <real expression> }
  \priority{ <non-negative integer> }
  \sojountimeLT{ <function> }
}

boolean expression = C/C++ boolean expression
```

Macro Name	Distribution	L.T. of corresponding pdf
exponential(λ,s);	$exp(\lambda)$	$\frac{\lambda}{s+\lambda}$
uniform(a,b,s);	$uni(a,b)$	$\frac{(e^{-as}-e^{-bs})}{(b-a)s}$
erlang(λ,n,s);	$erlang(\lambda,n)$	$\left(\frac{\lambda}{s+\lambda}\right)^n$ where n is an integer
gamma(λ,n,s);	$gamma(\lambda,n)$	$\left(\frac{\lambda}{s+\lambda}\right)^n$
deterministic(d,s);	$det(d)$	e^{-ds}
immediate();	$det(0)$	1 for all values of s
markov(s);	$exp(\mu_i)$ where μ_i is the sum of the rates of all enabled transitions in state i	$\frac{\mu_i}{s+\mu_i}$

Table 6.1. Some common transition delay density functions and their corresponding Laplace transforms.

```
real expression = C/C++ real expression
assignment = C/C++ assignment
function = C/C++ function returning a complex value
```

6.3.2 State-space Generator

Augmenting the input language in this fashion required modification to a number of components of DNAmaca. The parser had to be altered to recognise the new fields in the transition declarations, and extra methods were added to the `State` class to return the Laplace transforms of the transition firing delays. SMCA employs the same probabilistic state-space generator as DNAmaca, but instead of generating $\mathbf{Q}$ it is used to generate $\mathbf{P}$, the transition matrix of the model's EMC. Note that we do not generate the matrix $\mathbf{H}$ explicitly; instead it can be constructed when needed from $\mathbf{P}$ and the state sojourn time information parsed from the high-level model description.

6.3.3 Steady-state Solver

The steady-state solver also required a degree of reworking to calculate steady-state probabilities for SMPs. Eq. 2.4 states that in order to calculate these probabilities,

it is necessary to know the steady-state probabilities for the EMC and the average amount of time spent in each state. The former is calculated from $\mathbf{P}$ using the existing steady-state solution methods of DNAmaca. The latter is computed using the transition distribution information (parsed from the model file) of the transitions enabled in each state.

The average amount of time in a state is calculated as the sum of the mean firing times of the enabled transitions in that state, weighted with the probability that each transition fires. As our specification of the firing-time distributions of the transitions is linked to the demands of the numerical Laplace transform inversion algorithms, we make use of Eq. 3.12 to calculate the average time before a transition fires from the Laplace transform of its firing distribution. This is then used, along with the steady-state probabilities from the EMC, to solve for the steady-state probabilities of the SMP using Eq. 2.4. SMCA's performance analyser can then use these steady-state probabilities to calculate state and count measures in exactly the same way as DNAmaca.

6.3.4 Example SMP Steady-state Analysis

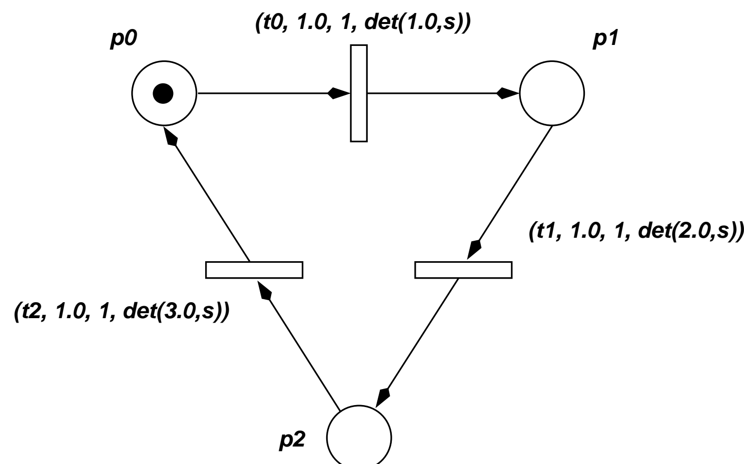

Fig. 6.5. A three-state SM-SPN.

We now demonstrate the analysis of a simple SMP using SMCA. Fig. 6.5 shows a three state SMP described as an SM-SPN, while Fig. 6.6 shows the corresponding SMCA input file. We also define four performance measures, in this case three state

```
\model{

  \statevector{ \type{short}{ p0, p1, p2 } }

  \initial{ p0 = 1; p1 = 0; p2 = 0; }

  \transition{t0}{
    \condition{p0 > 0}
    \action{ next->p0 = p0 - 1; next->p1 = p1 + 1; }
    \weight{1.0}
    \priority{1}
    \sojourntimeLT{ return deterministic(1.0, s); }
  }

  \transition{t1}{
    \condition{p1 > 0}
    \action{ next->p1 = p1 - 1; next->p2 = p2 + 1; }
    \weight{1.0}
    \priority{1}
    \sojourntimeLT{ return deterministic(2.0, s); }
  }

  \transition{t2}{
    \condition{p2 > 0}
    \action{ next->p2 = p2 - 1; next->p0 = p0 + 1; }
    \weight{1.0}
    \priority{1}
    \sojourntimeLT{ return deterministic(3.0, s); }
  }
}

\solution{ \method{ sor } }

\performance{
  \statemeasure{ mean_tokens_on_place_p0 }{
    \estimator{ mean }
    \expression{ p0 }
  }
  \statemeasure{ mean_tokens_on_place_p1 }{
    \estimator{ mean }
    \expression{ p1 }
  }
  \statemeasure{ mean_tokens_on_place_p2 }{
    \estimator{ mean }
    \expression{ p2 }
  }
  \countmeasure{ throughput_t2 }{
    \estimator{ mean }
    \transition{ t2 }
  }
}
```

Fig. 6.6. The SMCA input file for the SM-SPN in Fig. 6.5.

```
PerformanceAnalyser (0x8063248): powering up...
Timer (0x8063280): timer started...
Information (0x8063298): reading data from 'OPTIONS'.
Information (0x8063298): reading state data from 'INFO'.
PerformanceAnalyser (0x8063248): powered up...
Vector input (0x8063254) from 'SM-STEADY'
(precision set to 10 places)

(begin performance results)

State Measure 'mean_tokens_on_place_p0'

  mean                  1.6666666666e-01

State Measure 'mean_tokens_on_place_p1'

  mean                  3.3333333333e-01

State Measure 'mean_tokens_on_place_p2'

  mean                  5.0000000000e-01

Count Measure 'throughput_t2'

  mean                  1.6666666667e-01

(end performance results)
PerformanceAnalyser (0x8063248): powered down...
```

Fig. 6.7. The SMCA performance analyser output for the model file in Fig. 6.6.

measures which calculate the average number of tokens on each of the places and a count measure calculating the throughput of transition t_2. As there is only one token in the net, the state measure results correspond to the steady-state probabilities of being in each of the three states of the underlying SMP.

Fig. 6.7 shows the output from SMCA's performance analyser. Given the delays of the three deterministic transitions, we would expect that at steady-state the SMP would spend half its time in the state where t_2 is enabled, one third of the time in the state where t_1 is enabled and one sixth of the time time in the state where t_0 is enabled. In addition, transitions should fire once every 6 time units for a throughput of $1/6$. These intuitions are supported by the results from SMCA.

6.4 SMARTA

In this section we describe SMARTA (Semi-MArkov Passage Time Analyser), the final tool implementing work presented in this thesis. This draws upon technology from the semi-Markov steady-state solver SMCA and the Markovian passage time analyser HYDRA (both described above), as well as the GSPN analyser described in Chapter 3, and implements the iterative algorithm presented in Chapter 4 for the passage time analysis of very large semi-Markov models. As in HYDRA, hypergraph partitioning is employed to perform the central sparse matrix-vector multiplications efficiently in parallel. Many of the underlying techniques (probabilistic hash-based state exploration, efficient sparse matrix representation) are shared with HYDRA and SMCA, and so we concentrate here on novel features of the parallel Laplace transform inverter which implements our iterative algorithm.

6.4.1 Tool Architecture

The process of computing a passage time density using SMARTA is shown in Fig. 6.8 and begins with a high-level model specified in an enhanced form of the DNAmaca interface language. This input language combines the passage time measure spec-

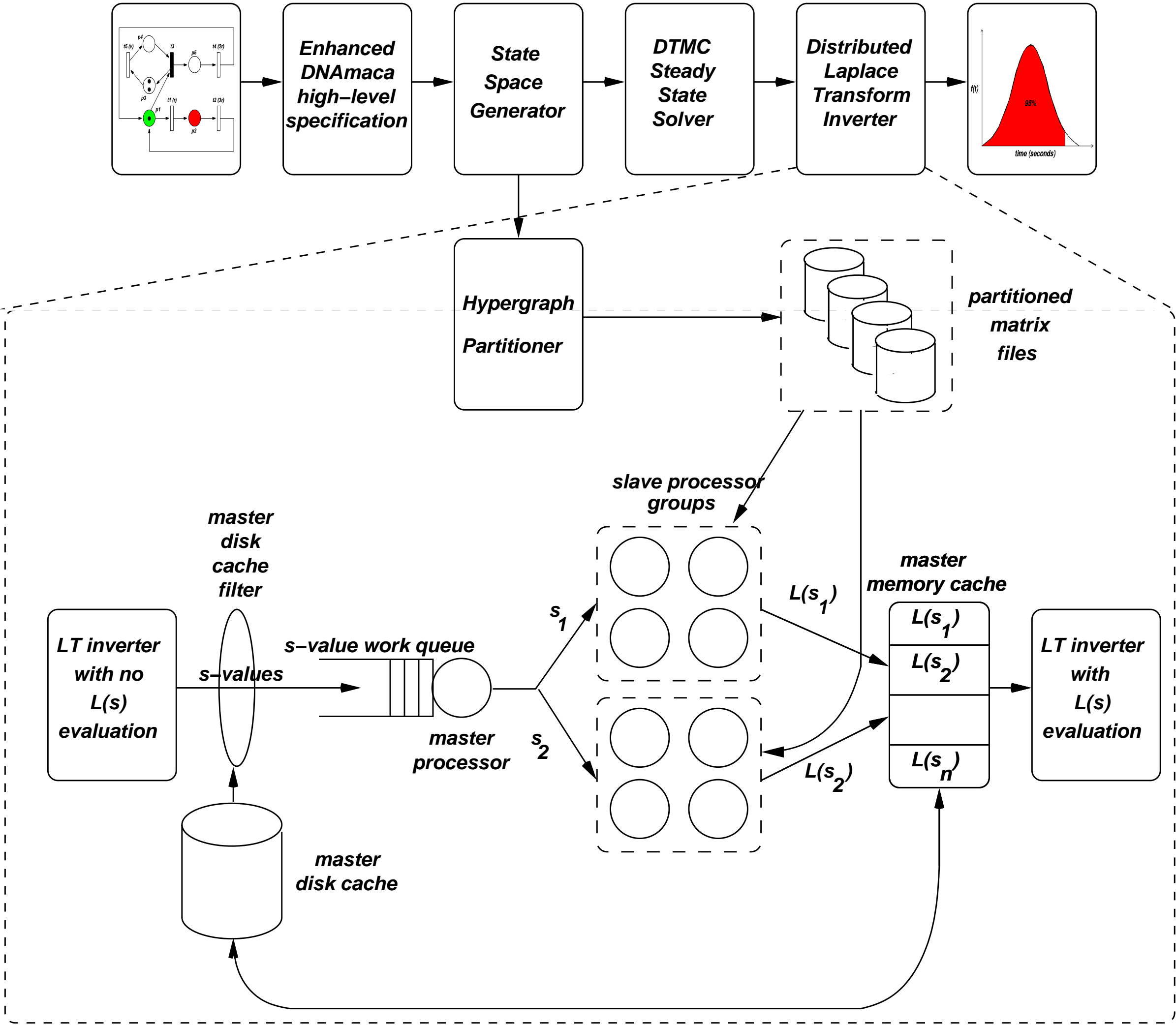

Fig. 6.8. SMARTA: Semi-Markov Passage Time Analyser.

ification syntax from HYDRA with SMCA's ability to specify generally distributed transitions. The model description is parsed into C/C++ source code and is compiled with SMCA's probabilistic, hash-based state generator and then executed to generate the state-space of the model's underlying semi-Markov process and the corresponding transition probability matrix $\mathbf{P}$ of the model's embedded Markov chain. A list of source and target states is also constructed. $\mathbf{U}$ and $\mathbf{U}'$ are then generated using $\mathbf{P}$ and the state holding-time distributions and target states specified in the high-level model. Normalised weights for the source states (the vector $\boldsymbol{\alpha}$ in Eqs. 3.5 and 4.5) are determined by the solution of $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$, as in HYDRA. The state-space is then partitioned using an off-the-shelf hypergraph partitioning tool. Finally, a distributed Laplace transform inverter (the operation of which is described in detail below) is used to calculate the value of the required passage time density or quantile at user-specified time-points. This can then be plotted using a program such as GNUplot.

6.4.2 Implementation of the Parallel Iterative Algorithm

The parallel passage time density calculator is implemented in C++ using the Message Passing Interface (MPI) standard [67] and employs a master-slave architecture with groups of slave processors. The master processor computes in advance the values of s at which it will need to know the value of $L_{\vec{i}\vec{j}}(s)$ in order to perform the inversion. As described in Section 4.1, this can be done irrespective of the inversion algorithm employed. The s-values are then placed in a global work-queue to which the groups of slave processors make requests.

The highest ranking processor in a group of slaves makes a request to the master for an s-value and is assigned the next one available. This is then broadcast to the other members of the slave group to allow them to construct their columns of the matrix $\mathbf{U}'$ for that specific s. Each processor reads in the columns of the matrix $\mathbf{P}$ that correspond to its allocated partition and uses the probabilities stored in these, along with the current value of s and the transition firing-time information from the parsed model file, to construct $\mathbf{U}'$ as described in Section 4.3.1. Also, each processor reads in the source-state weighting vector $\boldsymbol{\alpha}$.

```
begin
    y: vector of length |x_i|
    z_j: vector of length |x_j|, one for each j ∈ P, j ≠ i
    converged: boolean
    μ_i: complex number
    L(s): complex number
    x_i^T = α_t
    do begin
        for j = 0 to j = |P| − 1 do begin
            if j ≠ i
                irecv(j, z_j)
        end
        y^T = x_i^T U'_ii
        for j = 0 to j = |P| − 1 do begin
            if j ≠ i
                isend(j, y)
        end
        waitall(P)
        for j = 0 to j = |P| − 1 do begin
            if j ≠ i
                y^T = y^T + z_j^T U'_ij
        end
        for k = 0 to k = |x_i| − 1 do begin
            if y_k ∈ T
                μ_i += y_k
        end
        converged = true
        for k = 0 to k = |x_i| − 1 do begin
            if |y_k − (x_i)_k| > 10^-8 do begin
                converged = false
                break
            end
        end
        allreduce(converged, and, P)
        x_i^T = y^T
    end
    while not converged
    reduce(L(s), μ_i, sum, min(P), P)
    return L(s)
end
```

Fig. 6.9. Parallel iterative passage time calculation algorithm for slave processor i.

Fig. 6.9 outlines the iterative algorithm as implemented for slave processor i. P is the set of processors in a group of slaves and T is the set of target states. Each processor holds $|P|$ blocks of the matrix $\mathbf{U}'$ and we denote the jth block stored on processor i as $\mathbf{U}'_{ij}$, $0 \leq i, j, < |P|$. The block $\mathbf{U}'_{ii}$ (the on-diagonal block) is stored in the compressed sparse column format (cf. Section 6.1.8), while blocks $\mathbf{U}'_{ij}$, $i \neq j$, are stored in ultrasparse format (cf. Section 6.2.5). Note that the *block* $\mathbf{U}'_{ij}$ should not be confused with the (i, j)th *element* of $\mathbf{U}'$ denoted u'_{ij}. Each processor also maintains a portion of the initial weighting vector $\boldsymbol{\alpha}$ and the current iteration vector $\mathbf{x}$, and we denote the portion held by processor i as $\boldsymbol{\alpha}_i$ and $\mathbf{x}_i$ respectively.

Each processor in P also determines which vector elements need to be received from and sent to every other processor. A vector of vectors in which to store the non-zeros transmitted to it by the other processors in the group is then initialised. In Fig. 6.9, these are the $\mathbf{z}_p$ vectors. Only those elements required by remote processor j are sent from i, thus ensuring that a minimum amount of communication takes place. Processor i must therefore adjust the row indices in $\mathbf{U}'_{ij}$ so that they index into the vector of received elements $\mathbf{z}_j$. This makes multiplication of the $\mathbf{U}'_{ij}$ blocks with the $\mathbf{z}_j$ vectors efficient.

The iterative algorithm then proceeds until the difference between successive iteration vectors ($\mathbf{y}$ and $\mathbf{x}_i$) is less than some pre-specified amount (10^{-8} in Fig. 6.9) on all processors in P. On each iteration, processor i first sets up non-blocking receives to receive remote elements of $\mathbf{x}$ from the other processors. The operation `irecv(`$j, \mathbf{z}_j$`)` receives vector elements from processor j into the vector $\mathbf{z}_j$. Whilst waiting for these operations to complete, processor i multiplies its local matrix block $\mathbf{U}'_{ii}$ with its locally maintained portion of the vector $\mathbf{x}_i$ and the result stored in the vector $\mathbf{y}$. Processor i then sends the newly computed elements of $\mathbf{y}$ to the other processors which require them. The operation `isend(`$j, \mathbf{y}$`)` is a non-blocking operation which transmits those elements of $\mathbf{y}$ required by processor j to that processor.

Processor i then waits for the completion of non-blocking operations with a call to a `waitall(`P`)` function. This waits for all outstanding `isend()` and `irecv()` calls to processors in P involving i (either as source or destination) to complete be-

fore the algorithm proceeds. Remote vector elements stored in the $\mathbf{z}_j$ vectors are then multiplied with the $\mathbf{U}'_{ij}$ blocks and the result added to $\mathbf{y}$. Each processor then checks for convergence by comparing the absolute value of the difference between every element in the current and previous iteration vector. The algorithm has converged when the calculations of every slave in P has converged, and this is checked by performing an `allreduce()` with an and operator across all processors in P. This operation places the result of anding all the values of $converged$ together across P on every processor in P.

Once the calculations of a slave group are deemed to have converged, μ_j from all processors $j \in P$ are collected on the lowest-ranking processor using a reduce with the sum operator. In Fig. 6.9, this is the reduce($L(s)$, μ_j, sum, $\min(P)$, P) function, which accumulates the values of μ_j into the variable $L(s)$ on processor $\min(P)$ (the lowest-ranked processor $\in P$) from all processors in P. $L(s)$ is then returned to the global master by processor $\min(P)$ where it is cached. When all results have been computed and returned for all required values of s, the final Laplace inversion calculations are made by the master, resulting in the value of the passage time density or quantile at the required t-points. These points can then be displayed using a graph-plotting program such as GNUplot.

The next chapter will present examples of the analysis of very large semi-Markov models using SMARTA.

Chapter 7

Extended Continuous Stochastic Logic

Formal logics for stochastic systems provide a concise and rigorous way to pose performance questions and allow for the composition of simple queries into more complex ones. One such logic is Continuous Stochastic Logic (CSL), which was originally presented in [10, 11] and has been applied to Markovian state spaces in [13, 14, 85]. CSL can express performance measures by selecting states and paths from a system that meet both steady-state and passage time quantile criteria. CSL has been applied to Generalised Semi-Markov Processes [126] to specify performance properties on discrete event simulations, but its application to semi-Markov chains is relatively recent [98].

This chapter presents extended CSL (eCSL), which augments semi-Markov CSL with the ability to express a richer class of passage time quantities as well as measures based on transient state distributions. Unlike basic CSL, which operates at a state-transition level, eCSL is designed to operate at the model level on semi-Markov stochastic Petri nets (SM-SPNs, cf. Section 2.2.4), from which an underlying semi-Markov process can be generated. We first discuss the current state of CSL and highlight those areas which we enhance in eCSL. We then present eCSL and provide example formulae.

7.1 CSL

In order to make detailed comparisons with CSL, and to understand fully the enhancements introduced in eCSL, we first present a detailed summary of the standard CSL as used in [13, 14, 85, 98]. A semi-Markov CSL (similar to [98]) is defined directly over a semi-Markov state space $(S, \mathbf{P}, \mathbf{H}, A)$ where S is the set of states, $\mathbf{P}$ is the embedded probability transition matrix, $\mathbf{H}$ is the state holding time distribution matrix and A is a state labelling function. This labelling function attaches multiple labels to every state, and allows states to be associated with a more meaningful description (in terms of the high-level model) than solely their integer position in the transition matrix. For example, the label `failed` may be attached to several states; this label can then be used to refer to the states collectively rather than having to enumerate them explicitly as a set of state indices.

A CSL formula is defined as follows:

$$\Psi \overset{\text{def}}{=} \mathsf{tt} \;\Big|\; a \;\Big|\; \Psi \wedge \Psi \;\Big|\; \neg\Psi \;\Big|\; \mathcal{S}_{\boldsymbol{\rho}}(\Psi) \;\Big|\; \mathcal{P}_{\boldsymbol{\rho}}(\psi) \tag{7.1}$$

$$\psi \overset{\text{def}}{=} X\Psi \;\Big|\; \Psi \cup^{\boldsymbol{\tau}} \Psi \tag{7.2}$$

$\mathcal{S}$ represents a steady-state condition and $\mathcal{P}$ represents a passage time condition on a set of paths defined by ψ. The values $\boldsymbol{\rho}$ and $\boldsymbol{\tau}$ represent ranges of allowed probabilities and times, respectively.

The semantics of the logic are expressed by stating the precise conditions under which a single state s satisfies each of the possible clauses of a Ψ-formula. As in other temporal logics, this is written $s \models \Psi$.

The clause a is a label and a state s satisfies that label if $a \in A(s)$. Thus using the *not* and *conjunctive* clauses in combination with labelling allows whole sets of states to be defined with a Ψ-formula. The set of states specified in this manner is written $\mathrm{Sat}(\Psi) = \{s \in S \mid s \models \Psi\}$. Thus the steady-state clause $\mathcal{S}_{\boldsymbol{\rho}}(\Psi)$ defines a set of states $S_1 = \mathrm{Sat}(\Psi)$ and is true if the sum of the long-run probabilities of the states in S_1 lies in the range $\boldsymbol{\rho}$.

7.1.1 Formal CSL semantics

The formal semantics of CSL are:

$$\begin{aligned}
s &\models \mathsf{tt} && \forall s \\
s &\models a && \text{iff } a \in A(s) \\
s &\models \Psi_1 \wedge \Psi_2 && \text{iff } s \models \Psi_1 \wedge s \models \Psi_2 \\
s &\models \neg\Psi && \text{iff } s \not\models \Psi \\
s &\models \mathcal{S}_{\boldsymbol{\rho}}(\Psi) && \text{iff } \Pi_{\vec{j}} \in \boldsymbol{\rho} \text{ where } \vec{j} = \mathrm{Sat}(\Psi) \\
s &\models \mathcal{P}_{\boldsymbol{\rho}}(\psi) && \text{iff } \mathbb{P}\big(\sigma \in \mathrm{Path}(s) \mid \sigma \models \psi\big) \in \boldsymbol{\rho}
\end{aligned} \tag{7.3}$$

where $\mathrm{Path}(s)$ is the set of all paths starting from s. The quantity $\Pi_{\vec{j}}$ is the long term probability of being in any of the states in $\vec{j}$.

Further, a path σ satisfies a path formula, ψ, as follows:

$$\begin{aligned}
\sigma &\models X\Psi && \text{iff } \exists\sigma[1] \models \Psi \\
\sigma &\models \Psi_1 \cup^{\boldsymbol{\tau}} \Psi_2 && \text{iff } \exists u \in \boldsymbol{\tau} \,.\, (\sigma@u \models \Psi_2 \wedge \forall u' < u, \sigma@u' \models \Psi_1)
\end{aligned} \tag{7.4}$$

where $\sigma[1]$ is a state immediately succeeding the start state of σ and $\sigma@t$ is the state that the system is in at time t on the path σ. The X path operator is often referred to as the *next state operator* and is used to extract an aggregate DTMC probability for selecting a given set of successor states, $\mathrm{Sat}(\Psi)$. Finally, the *time-bounded until* formula, $\Psi_1 \cup^{\boldsymbol{\tau}} \Psi_2$, specifies a set of paths starting in a single state s which satisfy Ψ_1 for the duration of the path and terminate when they satisfy Ψ_2; this is further restricted to complete the passage in time $u \in \boldsymbol{\tau}$.

7.1.2 Opportunities for Enhancing CSL

There are three main issues regarding the specification of performance measures that arise from the definition of CSL:

1. Only a single start state can be specified for the time-bounded until formula with the existing formulation of CSL. Passage time specifications are more expressive when associated with many possible start states, especially when asking performance questions of high-level formalisms where the start and end conditions for a passage may not necessarily specify a unique state.

2. There is no ability to express performance conditions based on transient state distributions.

3. Although compound formulae of steady-state and passage time constraints are allowed, the meaning of the derived formulae is somewhat obscure.

As an example of this last point, consider $\mathcal{P}_{\boldsymbol{\rho}_1}\big(\mathcal{S}_{\boldsymbol{\rho}_2}(\Psi_1) \cup^{\boldsymbol{\tau}} \Psi_2\big)$ which would define a passage along a set of paths that consists of states which satisfy $\mathcal{S}_{\boldsymbol{\rho}_2}(\Psi_1)$, and terminate satisfying Ψ_2. As long as $\mathrm{Sat}(\Psi_1)$ has a steady-state value in $\boldsymbol{\rho}_2$ (and the underlying process is irreducible[†]), then all states will satisfy $\mathcal{S}_{\boldsymbol{\rho}_2}(\Psi_1)$, which therefore represents no constraint on the selected paths at all. Alternatively, if $\mathrm{Sat}(\Psi_1)$ does not have a steady-state value in $\boldsymbol{\rho}_2$, then only paths of length 0 may be selected. Similarly abstruse would be an $\mathcal{S}$ formula which relied on the possible start states of a $\mathcal{P}$ formula, for example, $\mathcal{S}_{\boldsymbol{\rho}_1}\big(\mathcal{P}_{\boldsymbol{\rho}_2}(\Psi_1 \cup^{\boldsymbol{\tau}} \Psi_2)\big)$: that is, calculating the long term state probability over the set of possible start states of the $\mathcal{P}$ portion.

7.2 eCSL

We present an extension to CSL, called eCSL, which can express a greater variety of performance-related questions: for example, transient state distribution-based properties and multiple start states for both passage and transient properties. Unlike standard CSL, which is applied directly to a labelled Markovian state space, eCSL operates on semi-Markov stochastic Petri nets, which can express any Markov or semi-Markov model (including those with immediate transitions). This eCSL is intended to complement CSL, as CSL concentrates on describing properties which are formally decidable while eCSL focuses on the specification of performance queries.

For the reasons given in Section 7.1.2, we remove the possibility of specifying compound formulae in the manner of CSL. We also simplify the path formulae of the

[†]The importance of this can be seen by considering an example with a state-space consisting of two strongly-connected components. The long-run probabilities will depend on which component is entered first, and if the target states only exist in one component then there is no possible path to them from the other.

original CSL and instead specify paths by providing high-level rules that yield a set of start states, a set of terminating states and a set of excluded states though which a path cannot pass. Taking into account the fact that CSL could not specify multiple start states, this is equivalently expressive to the logical until formula, which provides a single start state, a set of end states and a set of states that the passage is restricted to. In eCSL, sets of states themselves are specified in terms of markings of a semi-Markov stochastic Petri net. We believe that these simplifications make for a formalism which maps more pleasantly onto both Petri nets and the underlying stochastic quantities. It also keeps the path formulae required to specify complex performance measures simple.

7.2.1 The Syntax of eCSL

In this section, we define the syntax of eCSL over SM-SPNs. An eCSL statement Ψ acting on an SM-SPN system with set of markings M is defined by:

$$\Psi \stackrel{\text{def}}{=} \mathsf{tt} \;\Big|\; \Psi \wedge \Psi \;\Big|\; \neg\Psi \;\Big|\; \mathcal{S}_{\boldsymbol{\rho}}(\psi) \;\Big|\; \mathcal{T}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi, \psi) \;\Big|\; \mathcal{P}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi, \psi)$$

$$\psi \stackrel{\text{def}}{=} \mathsf{tt} \;\Big|\; p[N] \;\Big|\; \psi \wedge \psi \;\Big|\; \neg\psi$$

Here we have deliberately separated out the state specification ψ-formulae from the performance specification Ψ-formulae. This avoids the conceptual problems associated with the compound performance properties that arise in CSL, while still being sufficiently expressive to allow for multiple simultaneous performance criteria to be specified. We define the function which operates on a ψ-formula and extracts the set of all states that satisfy it as $\mathrm{Sat}(\psi) = \{m \in M \mid m \models \psi\}$.

In the ψ specification, $N \in \mathbb{N}_0$ and $p[N]$ is satisfied if the number of tokens on place p in some state m is in the set of allowed numbers of tokens N. As with CSL, $\boldsymbol{\rho} \in 2^{[0,1]}$ is a set of allowed probabilities and similarly $\boldsymbol{\tau} \in 2^{[0,\infty]}$ is a set of times. Here, 2^M is a power set which consists of all the subsets of the set M.

$\mathcal{S}_{\boldsymbol{\rho}}(\psi)$ is true if the steady-state probability of being in the set of states defined by ψ lies in the set $\boldsymbol{\rho}$.

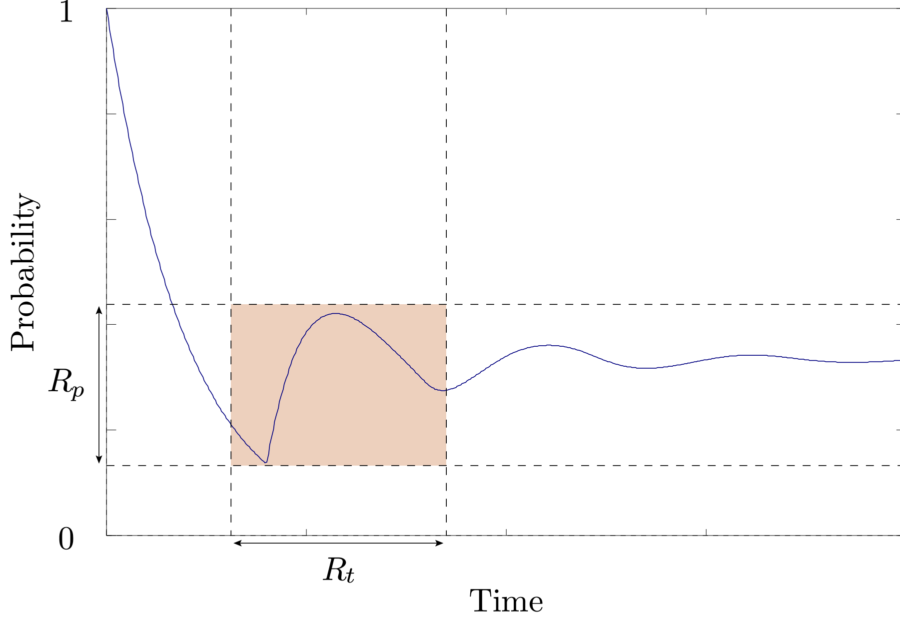

Fig. 7.1. An example of a transient constraint $\vec{m} \models \mathcal{T}_{R_p}^{R_t}(\Psi)$ which is satisfied by a transient distribution in the shaded area.

$\mathcal{T}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi_1, \psi_2)$ is satisfied by a set of start states if the probability of the system being in states $\mathrm{Sat}(\psi_1)$ at time t, while not having passed through states $\mathrm{Sat}(\psi_2)$, lies in $\boldsymbol{\rho}$ for all times $t \in \boldsymbol{\tau}$ (shown for an arbitrary transient distribution in Fig. 7.1).

Finally, $\mathcal{P}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi_1, \psi_2)$ is true for a set of start states if the random variable representing the passage time to target states $\mathrm{Sat}(\psi_1)$, while not having traversed states in $\mathrm{Sat}(\psi_2)$, lies in the range of times $\boldsymbol{\tau}$ with probability $p \in \boldsymbol{\rho}$. For a high-level modelling paradigm, we believe that specifying rules for sets of excluded states is simpler than having to specify explicitly all the permitted states for a path with state-by-state logical formulae as used by standard CSL on conventionally labelled state spaces.

7.2.2 Examples of eCSL Formulae

As an example of how eCSL could be used to pose performance questions in practice, we consider the problem of finding the value of q that satisfies the formula:

$$\mathrm{Sat}(p_1[35] \wedge p_5[10]) \models \mathcal{P}_{\{q\}}^{[0,10)}(p_2[175], p_6[1]) \tag{7.5}$$

The question being asked is: what is the probability that a defined passage takes less than time 10? The passage time quantity is defined by the source states $p_1[35] \wedge p_5[10]$, by the target states $p_2[175]$ and taking into account the excluded states, $p_6[1]$. These expressions define sets of states, for instance $p_1[35] \wedge p_5[10]$ selects all the Petri net markings which have 35 tokens on p_1 and 10 tokens on p_5.

If we wish to define multiple performance requirements for a single set of start states on a system then we might ask:

$$\mathrm{Sat}(p_1[35] \wedge p_5[10]) \models \left(\mathcal{P}^{[0,10)}_{(0.9,1]}(p_2[175], p_6[1]) \wedge \mathcal{P}^{[0,100)}_{(0.98,1]}(p_2[320], p_6[4])\right) \tag{7.6}$$

This expresses the need to surpass 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 100 time units. In this way, multiple quality of service requirements may be succinctly expressed and verified with a single eCSL formula.

If distinct start states are required for each performance measure, we could compose them as follows:

$$\left(\mathrm{Sat}(p_5[10]) \models \mathcal{P}^{[0,10)}_{(0.9,1]}(p_2[175], p_6[1])\right) \wedge \left(\mathrm{Sat}(p_1[35]) \models \mathcal{T}^{[0,100)}_{(0.2,1]}(p_2[320], p_6[4])\right) \tag{7.7}$$

This expresses the need to surpass a 90% quantile for a passage time within the first 10 time units starting from states where there are 10 tokens on p_5, and a 20% quantile for a passage time within the first 100 time units starting from states where there are 35 tokens on p_1.

7.2.3 Formal Stochastic Semantics of eCSL

In this section, we formally define the satisfiability formulae for eCSL over SM-SPNs. These are expressed in terms of a marking m of an SM-SPN where $m(p)$ is the number of tokens at place p in marking m. We test individual markings of the Petri net against every allowed combination of Ψ and ψ-expressions. As before, these are evaluated in terms of individual satisfiability questions, e.g. $m \models \psi_1$, which poses the question: does the single state m satisfy the formula ψ_1?

Formally, for the general ψ-expression:

$$\begin{aligned} m &\models \text{tt} && \forall m \\ m &\models p[N] && \text{iff } m(p) \in N \\ m &\models \psi_1 \wedge \psi_2 && \text{iff } m \models \psi_1 \wedge m \models \psi_2 \\ m &\models \neg\psi && \text{iff } m \not\models \psi \end{aligned} \tag{7.8}$$

Importantly, the Ψ-formulae are satisfied by vectors of markings or states. This is so that a configuration of multiple start states can be defined and used to specify corresponding multiple performance properties. This overcomes the restriction inherent in CSL that it is only able to express performance properties with single start states.

$$\begin{aligned} \vec{m} &\models \text{tt} && \forall \vec{m} \\ \vec{m} &\models \Psi_1 \wedge \Psi_2 && \text{iff } \vec{m} \models \Psi_1 \wedge \vec{m} \models \Psi_2 \\ \vec{m} &\models \neg\Psi && \text{iff } \vec{m} \not\models \Psi \\ \vec{m} &\models \mathcal{S}_{\boldsymbol{\rho}}(\psi) && \text{iff } \Pi_{\vec{j}} \in \boldsymbol{\rho} \text{ where } \vec{j} = \mathrm{Sat}(\psi) \\ \vec{m} &\models \mathcal{T}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi_1, \psi_2) && \text{iff } \forall t \in \boldsymbol{\tau}, T_{\vec{m}\vec{j}}^{\vec{k}}(t) \in \boldsymbol{\rho} \text{ where} \\ & && \quad \vec{j} = \mathrm{Sat}(\psi_1), \vec{k} = \mathrm{Sat}(\psi_2) \\ \vec{m} &\models \mathcal{P}_{\boldsymbol{\rho}}^{\boldsymbol{\tau}}(\psi_1, \psi_2) && \text{iff } \mathbb{P}\big(P_{\vec{m}\vec{j}}^{\vec{k}} \in \boldsymbol{\tau}\big) \in \boldsymbol{\rho} \text{ where} \\ & && \quad \vec{j} = \mathrm{Sat}(\psi_1), \vec{k} = \mathrm{Sat}(\psi_2) \end{aligned} \tag{7.9}$$

The steady-state operator $\Pi_{\vec{j}}$ represents the long term probability of being in the set of states $\vec{j}$ and is independent of any start state (assuming that the underlying SMP is irreducible).

For the transient operator $\mathcal{T}$ we have a modified transient distribution function to take account of the excluded states in $\vec{k}$:

$$T_{\vec{i}\vec{j}}^{\vec{k}}(t) = \sum_{i \in \vec{i}} \alpha_i \mathbb{P}\big(Z(t) \in \vec{j} \mid Z(0) = i, \forall t' < t \,.\, Z(t') \notin \vec{k}\big) \tag{7.10}$$

The $\mathbb{P}(\cdot)$ term inside the summation describes the conditional probability that the SMP is in a state in $\vec{j}$ at time t given that it started from a state i and has never been through any state in $\vec{k}$. This probability is finally deconditioned over the set of all the possible start states in $\vec{i}$. $\boldsymbol{\alpha}$ is taken to be a normalised steady-state vector, but there is no reason why it could not be generalised to an arbitrary initial weighting vector, specified by the user (although there is currently no syntactic support for this in eCSL).

Similarly for the passage time operator, $\mathcal{P}$, we can modify the passage time random variable to incorporate the excluded states vector $\vec{k}$:

$$P_{\vec{i}\vec{j}}^{\vec{k}} = \sum_{i \in \vec{i}} \alpha_i \inf\{u > 0 : Z(u) \in \vec{j} \mid Z(0) = i, \forall u' < u \, . \, Z(u') \notin \vec{k}\} \tag{7.11}$$

Calculating the modified passage time probability $\mathbb{P}\big(P_{\vec{m}\vec{j}}^{\vec{k}} \in \boldsymbol{\tau}\big) \in \boldsymbol{\rho}$ or transient probability $T_{\vec{i}\vec{j}}^{\vec{k}}(t)$ quantities involves straightforward modification of the formulae of the standard passage time and transient formulae for an SMP given in Chapter 4. All the excluded states in an exclusion set $\vec{k}$ are removed from the embedded DTMC ($\forall i \in S, k \in \vec{k}$ let $p_{ik} = 0$ and $p_{ki} = 0$), while renormalising the probabilities as necessary, so that $\sum_j p_{ij} = 1$ for all i. The renormalising of the DTMC, after removal of the excluded states, reflects the conditional nature of Eq. 7.10 and Eq. 7.11.

More examples of eCSL formulae are given in Section 8.2 of the next chapter.

Chapter 8

Numerical Results for Very Large Markov and Semi-Markov Models

This chapter presents passage time and transient results calculated in very large Markov and semi-Markov models using the tools described in Chapter 6. This serves as a demonstration of the capacity of our methods and of the types of performance queries they can answer. We first analyse two Markov models with up to 10.8 million states using HYDRA. Numerical passage time results are presented and are validated against simulation. We also examine the scalability of HYDRA on two parallel architectures.

We then demonstrate the analysis of very large semi-Markov models using SMARTA. We begin by conducting transient analysis on a $106\,540$ state model, and then move onto the passage time analysis of models with up to 15.4 million states. Both passage time density and quantile results are produced. We also demonstrate the use of eCSL for the formal specification of performance queries. Finally, the scalability of SMARTA is investigated on the Viking Beowulf cluster.

8.1 Very Large Markov Models

In this section we present passage time results for the analysis of very large Markov models using HYDRA. We consider two models: the Flexible Manufacturing System

GSPN described in Appendix A.2 and the tree-like queueing network of Appendix A.3. We produce passage time results for the FMS model with $1\,639\,440$ states and for the tree-like queueing network with $10\,874\,304$ states, and validate them against simulation and a known analytical solution respectively. In the case of the tree-like queueing network, we also present figures for the number of non-zeros and amount of data exchanged after each iteration under a number of partitioning schemes. The scalability of HYDRA is investigated on two architectures (a dedicated parallel computer and a network of PC workstations).

8.1.1 Flexible Manufacturing System

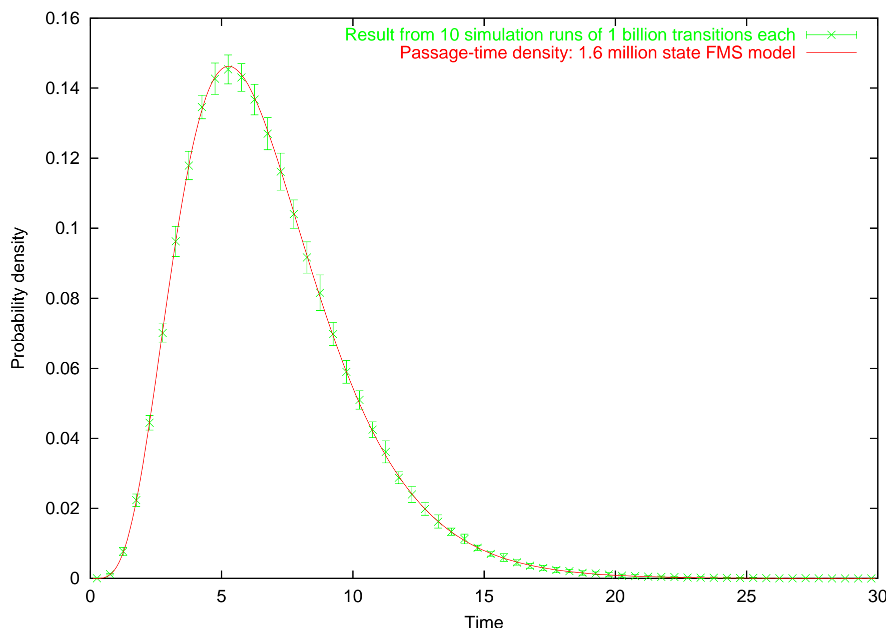

Fig. 8.1. Numerical and simulated (with 95% confidence intervals) passage time densities for the time taken to produce a finished part of type $P12$ starting from states in which there are $k = 7$ unprocessed parts of types $P1$ and $P2$.

The first Markov model for which we present results is the Flexible Manufacturing System GSPN. For $k = 7$, the GSPN's underlying Markov chain has $1\,639\,440$ states and $13\,552\,968$ non-zero off-diagonal entries in its generator matrix $\mathbf{Q}$. For this model, we calculate the density of the time taken to produce a finished part of type $P12$ starting from any state in which there are 7 unprocessed parts of type $P1$ and 7 unprocessed

parts of type $P2$. That is, the source markings (of which there are 36, corresponding to the possible submarkings of $M3$) are those where $M(P1) = M(P2) = 7$ and the target markings (of which there are 429 624) are those where $M(P12s) = 1$. We weight the density from each source state according to the relative probability that the passage originates in that state (cf. Eq. 3.10).

After modification of the state graph to allow for transitions from target states to a new terminal state, the uniformized matrix $\mathbf{P}'$ has $11\,001\,408$ non-zero entries. The hypergraph tool PaToH is then used to partition the rows of the transposed matrix $\mathbf{P}'^T$ as input to our parallel algorithm. Fig. 8.1 shows the resulting numerically calculated passage time density, which is validated against the combined results from 10 simulations (each of which consisted of 1 billion transition firings) plotted with 95% confidence bounds. There is excellent agreement between the numerical and simulated passage time densities.

8.1.2 Tree-like Queueing Network

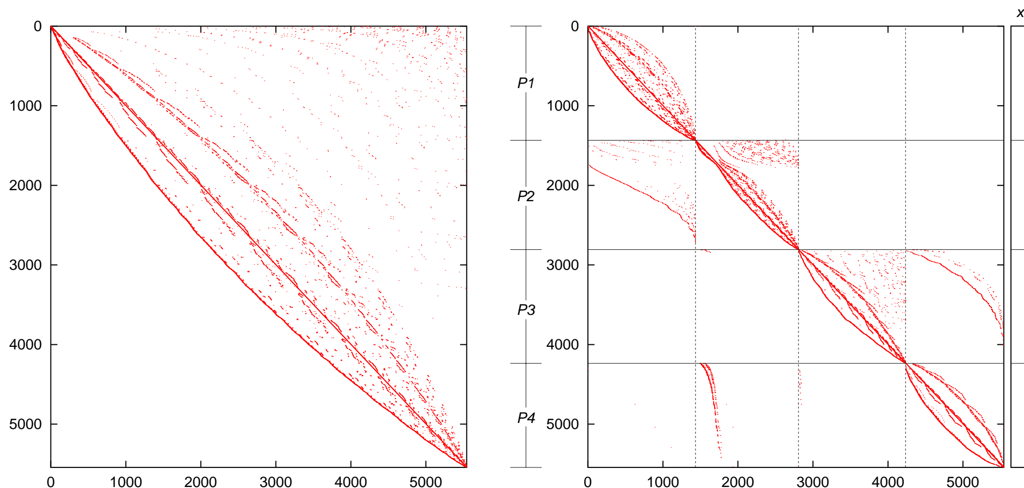

Fig. 8.2. Transposed $\mathbf{P}'$ matrix (left) and hypergraph-partitioned matrix (right) for the tree-like queueing network with 6 customers (5 544 states).

The second example we consider is a cycle time in the closed tree-like queueing network. For a small six-customer system with 5 544 states, Fig. 8.2 shows the resulting

proc-essor	non-zeros	local %	remote %	reused %
1	7 022	99.96	0.04	0
2	7 304	91.41	8.59	34.93
3	6 802	88.44	11.56	42.11
4	6 967	89.01	10.99	74.28

	1	2	3	4
1	-	407	-	4
2	3	-	16	181
3	-	-	-	12
4	-	1	439	-

Table 8.1. Communication overhead in the queueing network model with six customers (left) and interprocessor communication matrix (right) for each processor in a 4 processor decomposition.

Partitioning Method	Communication Overhead	
	Messages	Volume (MB)
randomised	240	450.2
linear	134	78.6
graph-based	110	19.7

Table 8.2. Per-iteration communication overhead for various partitioning methods for the queueing network model with 27 customers on 16 processors.

transposed $\mathbf{P}'$ matrix and associated hypergraph decomposition produced by hMETIS for a 4 processor decomposition. Statistics about the per-iteration communication associated with this decomposition are presented in Table 8.1. Around 90% of the non-zero elements allocated to each processor are local, i.e. they are multiplied with vector elements that are stored locally. The remote non-zero elements are multiplied with vector elements that are sent from other processors. However, because the hypergraph decomposition tends to align remote non-zero elements in columns (well illustrated in the 2nd block belonging to processor 4), reuse of received vector elements is good (up to 74%) with correspondingly lower communication overhead. The communication matrix on the right in Table 8.1 shows the number of vector elements sent between each pair of processors during each iteration (e.g. 181 vector elements are sent from processor 2 to processor 4).

Moving to a more sizeable model, the queueing network with 27 customers has an un-

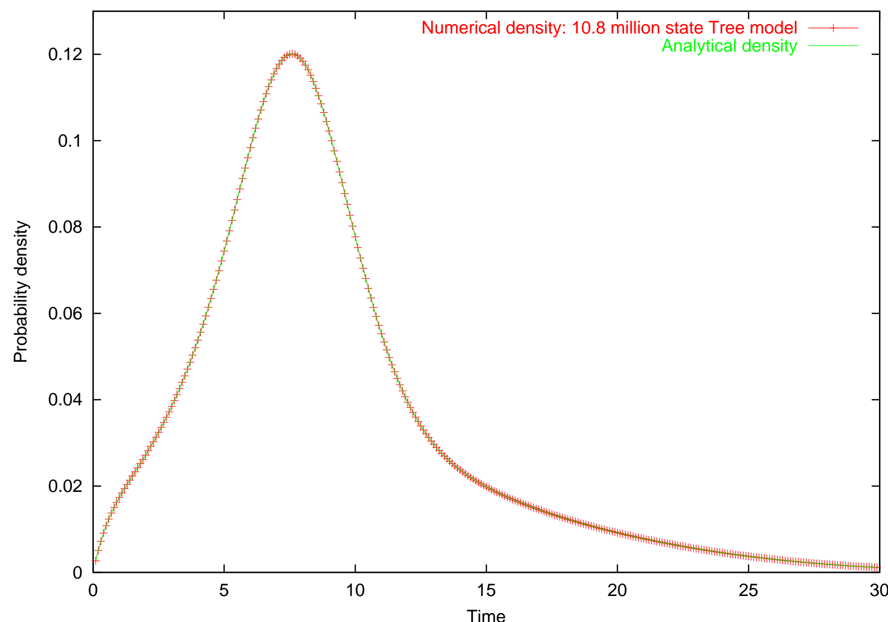

Fig. 8.3. Numerical and analytical cycle time densities for the tree-like queueing network of Fig. A.3 with 27 customers (10 874 304 states).

derlying Markov Chain with 10 874 304 states and 82 883 682 transitions. This model is too large to partition using a hypergraph partitioner on a single machine (even one with 2GB RAM), and consequently a lesser quality graph-based decomposition produced by the parallel graph partitioner ParMETIS (running on the PC cluster) was used. The options chosen were to use the parallel partitioning algorithm, a successive folding level of 300 [84] and weights on both vertices and edges. It must be noted that this decomposition still offers a great reduction in communication costs over other methods available: a 16-way partition has an average of 95.8% local non-zero elements allocated to each processor and a reused received non-zero element average of 30.4%. Table 8.2 shows the per-iteration communication overhead for randomised (i.e. random assignment of rows to partitions), linear (i.e. simple in-order allocation of rows to processors such that the number of non-zeros assigned to each processor is the same) and graph-based allocations. The graph-based method is clearly superior, both in terms of number of messages sent and (especially) communication volume.

Fig. 8.3 compares the numerical and analytical cycle time densities for the queueing network with 27 customers. Readers are directed to Appendix A.3 for details of the

p	AP3000			PC Cluster			Comm. per iteration	
	time (s)	S_p	E_p	time (s)	S_p	E_p	Messages	Vol (MB)
1	1243.3	1.00	1.000	325.0	1.00	1.000	0	0
2	630.5	1.97	0.986	258.7	1.26	0.628	2	1.5
4	328.2	3.78	0.947	197.1	1.65	0.412	12	3.2
8	182.3	6.82	0.853	143.0	2.27	0.284	51	5.3
16	99.7	12.47	0.779	114.6	2.84	0.178	207	7.3
32	58.6	21.22	0.663	71.7	4.53	0.142	663	9.6

Table 8.3. Run-time, speedup (S_p), efficiency (E_p) and per-iteration communication overhead for p-processor passage time density calculation in the FMS model with $k = 7$. Results are presented for an AP3000 distributed-memory parallel computer and a PC cluster.

analytical solution. Agreement is excellent and the results agree to an accuracy of 0.00001% over the time range plotted. The numerical density is computed in 968 seconds (16 minutes 8 seconds) for 875 iterations using 16 Athlon 1.4GHz PCs with 512MB RAM linked together by 100Mbps switched Ethernet. The memory used on each PC is just 84MB. It was not possible to compute the density on a single PC (with 512MB RAM) but the same computation on a dual-processor server machine (with 2GB RAM) required $5\,580$ seconds (93 minutes).

8.1.3 HYDRA Scalability

Table 8.3 shows the performance of our algorithm on two architectures: a Fujitsu AP3000 distributed-memory parallel computer running Solaris and a Linux-based PC workstation cluster. The AP3000 is based on a grid of 60 processing nodes, each of which has a UltraSPARC 300MHz processor and 256MB RAM. These nodes are interconnected by a 2D wraparound mesh network that uses wormhole routing and that has a peak throughput of 520Mbps. The PC cluster is a vanilla network of workstations, consisting of 32 Athlon 1.4GHz PCs each with 512MB RAM linked together by a 100Mbps switched Ethernet network. Distributed run-time is measured as the maximum processor run time from the start of the first uniformization iteration. The

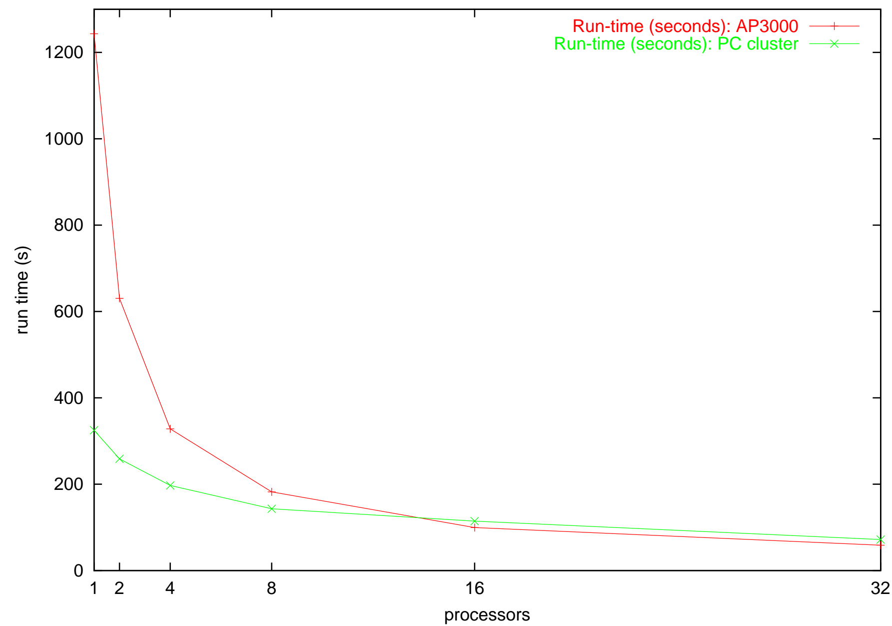

Fig. 8.4. Distributed run-time for the FMS model with $k = 7$ on the AP3000 and a PC cluster.

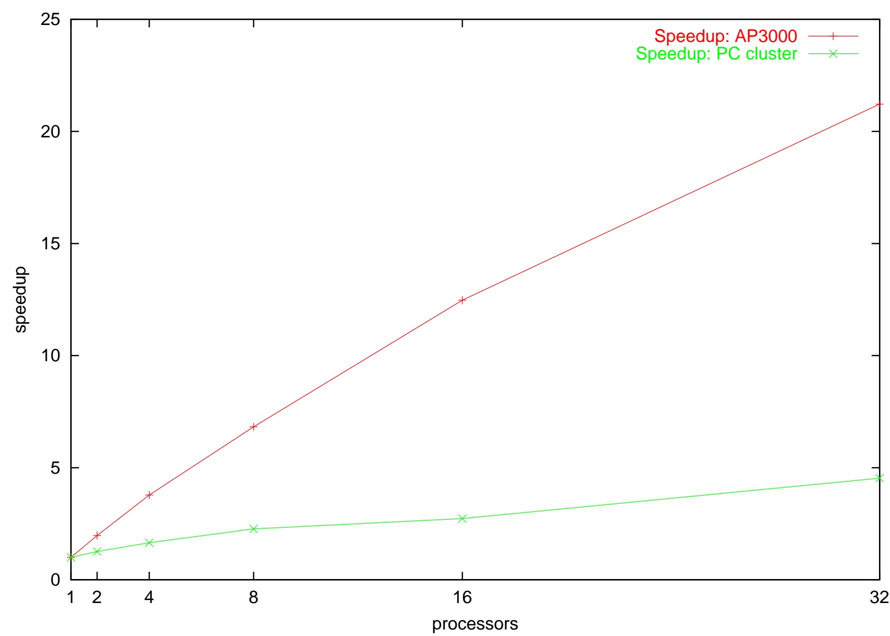

Fig. 8.5. Speedup for the FMS model with $k = 7$ on the AP3000 and a PC cluster.

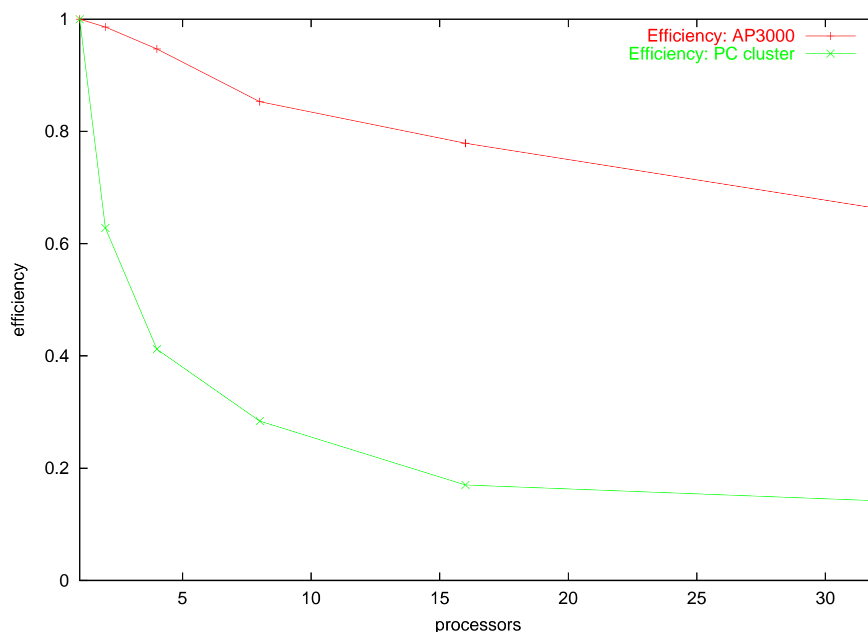

Fig. 8.6. Efficiency for the FMS model with $k = 7$ on the AP3000 and a PC cluster.

speedup for p processors, denoted by S_p, is given by the run time of the sequential solution ($p = 1$) divided by the run time with p processors. Efficiency for p processors, denoted by E_p, is defined as $E_p = S_p/p$. In every case, the sparse matrix was partitioned using PaToH with the same partitioning options as in Section 5.1.3. The machine used for partitioning was an Intel Pentium 4 2.6GHz machine with 1GB of RAM.

Corresponding graphs of the run-time, speedup and efficiency on each architecture are presented in Fig. 8.4, Fig. 8.5 and Fig. 8.6. The speedups and efficiencies achieved on the AP3000 are excellent. Solution time on a single AP3000 node is 20 minutes 43 seconds whereas on 32 processors it takes just 58.6 seconds (i.e. 21.22 times faster, corresponding to an efficiency of 66.3%).

With processors that are about 4 times faster and a communication network that is about 6 times slower than the AP3000, and without exclusive access to either processors or the interconnection network, we cannot expect such good results on the (shared departmental) PC cluster. However, unusually for problems of this type, reasonable speedups are still achieved, requiring 5 minutes 25 seconds on a single PC and

1 minute 12 seconds on 32 PCs (i.e. 4.53 times faster, corresponding to an efficiency of 14.2%). The speedup trend for the PC cluster is shallow but linear, suggesting that speedup will continue to improve for an even larger number of processors. Adding extra workstations also boosts solution capacity through additional RAM. Note that the results presented for the PC cluster were gathered at times when the network and processors were most likely to be idle (e.g. late at night) and have been averaged over three runs to minimise the impact of any external interference.

8.2 Very Large Semi-Markov Models

In this section, we display transient state distributions and passage time densities produced from semi-Markov models using SMARTA. We analyse two models: the Voting model of Appendix A.4 and the Web-server model of Appendix A.5. State spaces of up to 15.4 million states are analysed and the results are validated by simulation. Examples of eCSL queries for these measures are provided. We also present scalability results for SMARTA.

The results presented in this section were produced on the Viking Beowulf Linux cluster with 64 dual-processor nodes. Each node has two Intel Xeon 2.0GHz processors and 2GB of RAM. The nodes are connected by a Myrinet network with a peak throughput of 2Gbps.

8.2.1 Transient Analysis

Fig. 8.7 shows the transient state distribution for the transit of five voters from place p_1 to p_2 in system 3 ($106\,540$ states) of the Voting model. We can pose performance questions about the transient behaviour of the system using eCSL, for example:

$$\mathrm{Sat}(p_1[60] \wedge p_3[25] \wedge p_5[4]) \models \mathcal{T}^{(0,50]}_{[0,0.18)}(p_2[5])$$

This asks the question: does the probability of having processed exactly 5 voters (represented by the movement of 5 tokens from p_1 to p_2) stay below 0.18 for the first 50

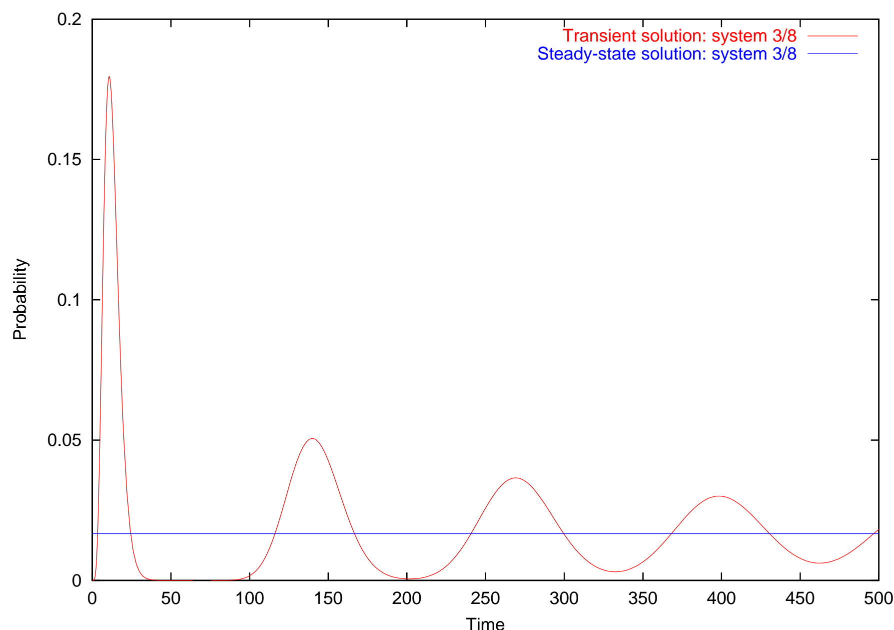

Fig. 8.7. Transient and steady-state values in Voting system 3, for the transit of 5 voters from the initial marking to place p_2.

seconds of the operation of the system? By inspection of Fig. 8.7, we can see that it does.

This graph can be compared with the results from the much smaller example shown in Fig. 4.15. There is a more noticeable separation between the first two peaks in Fig. 8.7 as there are many more voters to be processed (60 rather than 18).

8.2.2 Passage Time Analysis

Fig. 8.8 shows the density of the time taken to process 300 voters (as given by the passage of 300 tokens from place p_1 to p_2) in system 8 ($10\,991\,400$ states) of the Voting model. Numerical calculation of the density required 15 hours and 7 minutes using 64 slave processors (in 8 groups of 8) for the 31 t-points plotted. Our algorithm evaluated $L_{\vec{i}\vec{j}}(s)$ at $1\,023$ s-points, each of which involved manipulating sparse matrices of rank $10\,999\,140$. The curve is validated against the combined results from 10 simulations, each of which consisted of 1 billion transition firings. Despite this large simulation effort, we still observe wide confidence intervals (probably because of the rarity of source states).

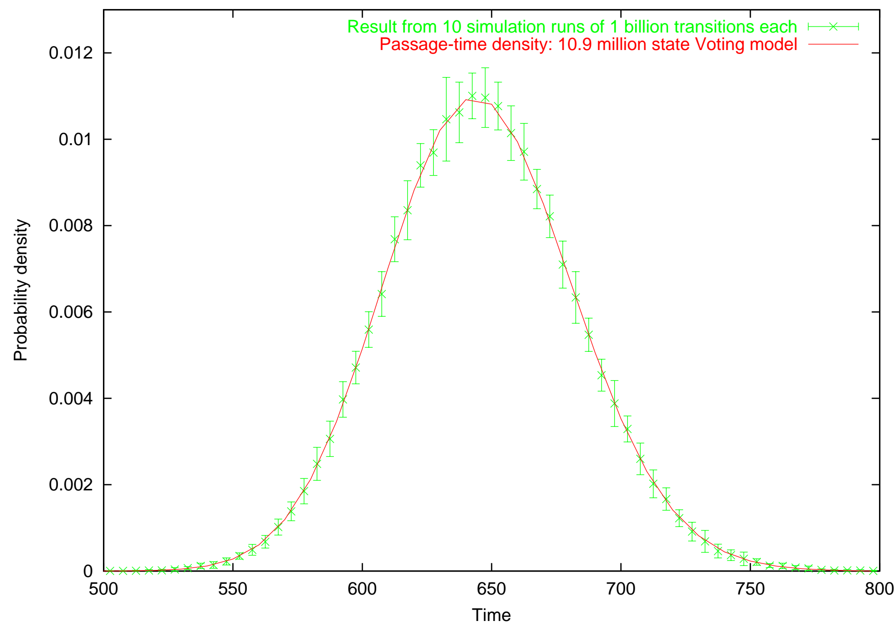

Fig. 8.8. Numerical and simulated (with 95% confidence intervals) density for the time taken to process 300 voters in the Voting model system 8 (10.9 million states).

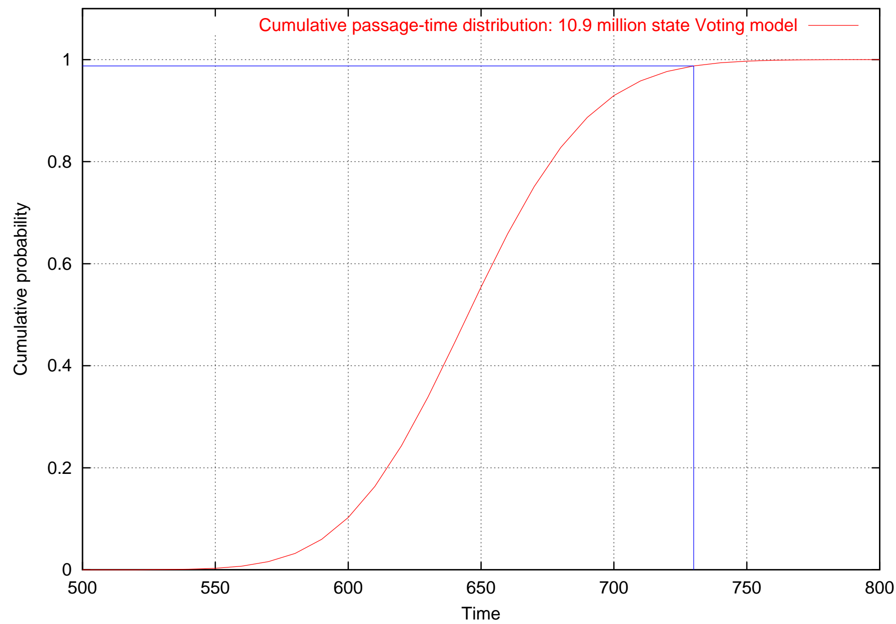

Fig. 8.9. Cumulative distribution function and quantile of the time taken to process 300 voters in the Voting model system 8 (10.9 million states).

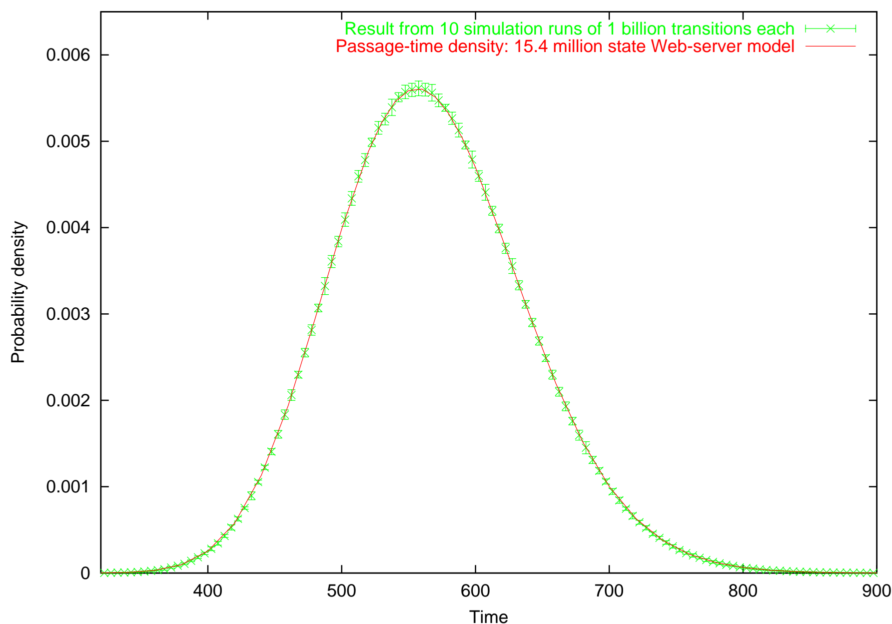

Fig. 8.10. Numerical and simulated (with 95% confidence intervals) density for the time taken to process 100 reads and 50 page updates in the Web-server model system 6 (15.4 million states).

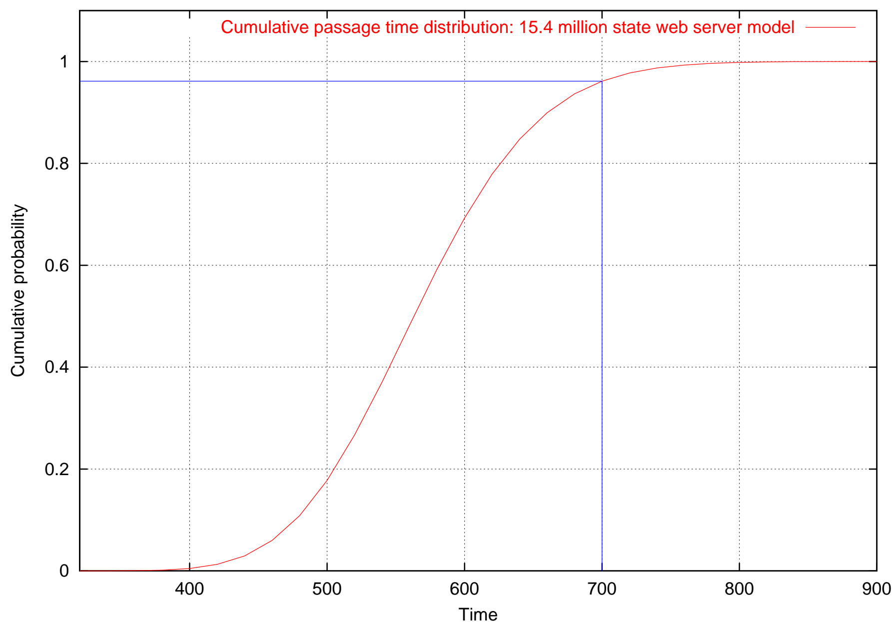

Fig. 8.11. Cumulative distribution function and quantile of the time taken to process 100 reads and 50 page updates in the Web-server model system 6 (15.4 million states).

Fig. 8.9 is a cumulative distribution for the same passage as Fig. 8.8, which again allows us to extract reliability quantiles. For instance:

$$\mathbb{P}(\text{system 8 can process 300 voters in less than 730 seconds}) = 0.9876$$

In eCSL, this query would satisfy:

$$\mathrm{Sat}(p_1[300] \wedge p_3[80] \wedge p_5[10]) \models \mathcal{P}^{[0,730)}_{(0.98,1]}(p_2[300])$$

Fig. 8.10 shows the density of the time taken to perform 100 reads and 50 page updates in the Web-server model 6 ($15\,445\,919$ states). Calculation of the 35 t-points plotted required 2 days, 17 hours and 30 minutes using 64 slave processors (in 8 groups of 8). Our algorithm evaluated $L_{\vec{ij}}(s)$ at $1\,155$ s-points, each of which involved manipulating sparse matrices of rank $15\,445\,919$. Again, the numerical result is validated against the combined results from 10 simulations, each of which consisted of 1 billion transition firings. We observe excellent agreement. Fig. 8.11 shows the corresponding cumulative distribution function with a reliability quantile superimposed, in this case showing:

$$\mathbb{P}(\text{system 6 can process 100 reads and 50 page updates in less than 700 seconds}) = 0.9613$$

In eCSL, this query would satisfy:

$$\mathrm{Sat}(p_1[100] \wedge p_8[50]) \models \mathcal{P}^{[0,700)}_{(0.96,1]}(p_2[100] \wedge p_9[50])$$

8.2.3 SMARTA Scalability

Table 8.4 and Figs. 8.12, 8.13 and 8.14 display the performance of the hypergraph-partitioned sparse matrix–vector multiplication operations on the Viking Beowulf cluster. They show good scalability with a linear speedup trend, which is unusual in problems of this nature. This is because the hypergraph partitioning minimises the amount of data which must be exchanged between processors. The efficiency is not 100% in all cases, however, as even this reduced amount of inter-node communication imposes an overhead and computational load is not perfectly balanced.

Processors	Time (s)	Speedup	Efficiency
1	3968.07	1.00	1.00
2	2199.98	1.80	0.902
4	1122.97	3.53	0.883
8	594.07	6.68	0.835
16	320.19	12.39	0.775
32	188.14	21.09	0.659

Table 8.4. Run-time, speedup and efficiency of performing hypergraph-partitioned sparse matrix–vector multiplication across 1 to 32 processors. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster.

Processors	Time (s)	Speedup	Efficiency
32×1	150.13	26.43	0.830
16×2	159.55	24.87	0.777
8×4	162.13	24.47	0.765
4×8	165.24	24.01	0.750
2×16	173.76	22.84	0.714
1×32	188.14	21.09	0.659

Table 8.5. Run-time, speedup and efficiency using 32 slave processors divided into various different size sub-clusters. Calculated for the $249\,760$ state Voting model for 165 s-points on the Viking Beowulf cluster.

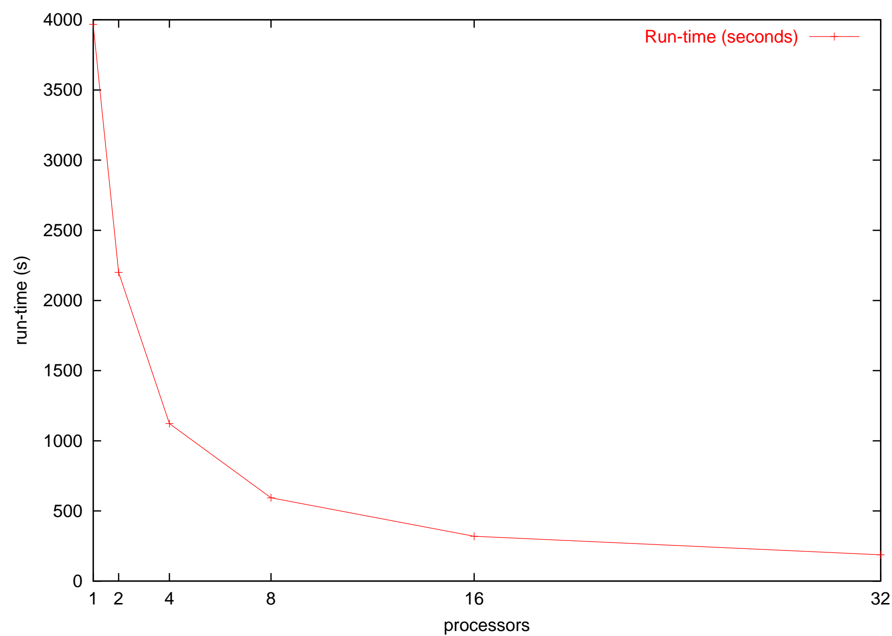

Fig. 8.12. Run-time of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the 249 760 state Voting model for 165 s-points on the Viking Beowulf cluster.

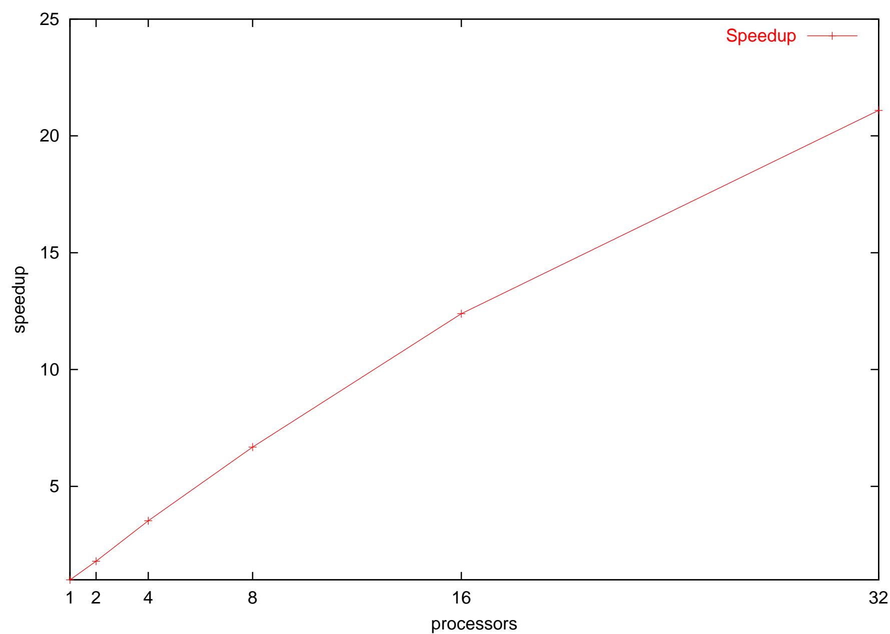

Fig. 8.13. Speedup of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the 249 760 state Voting model for 165 s-points on the Viking Beowulf cluster.

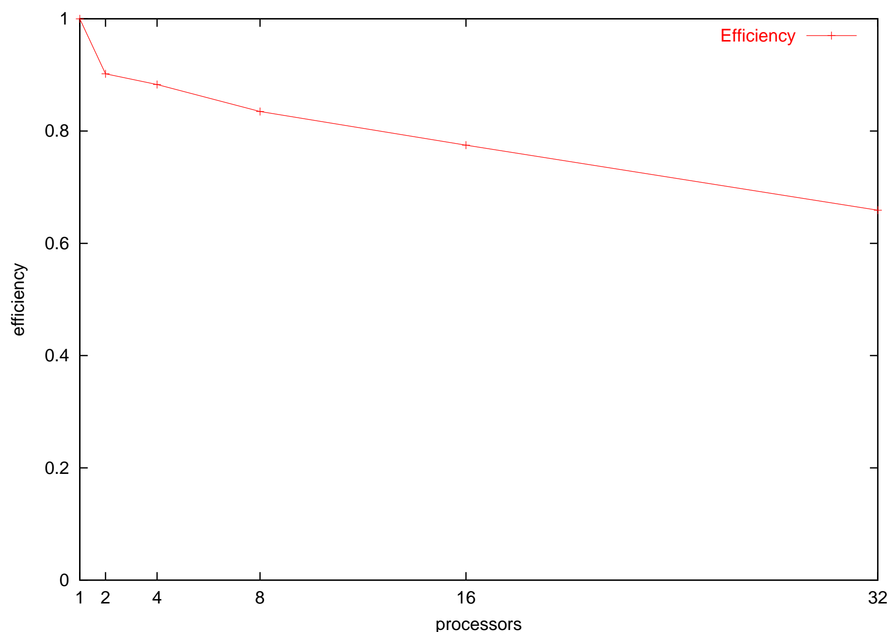

Fig. 8.14. Efficiency of hypergraph-partitioned sparse matrix–vector multiplication. Calculated for the 249 760 state Voting model for 165 s-points on the Viking Beowulf cluster.

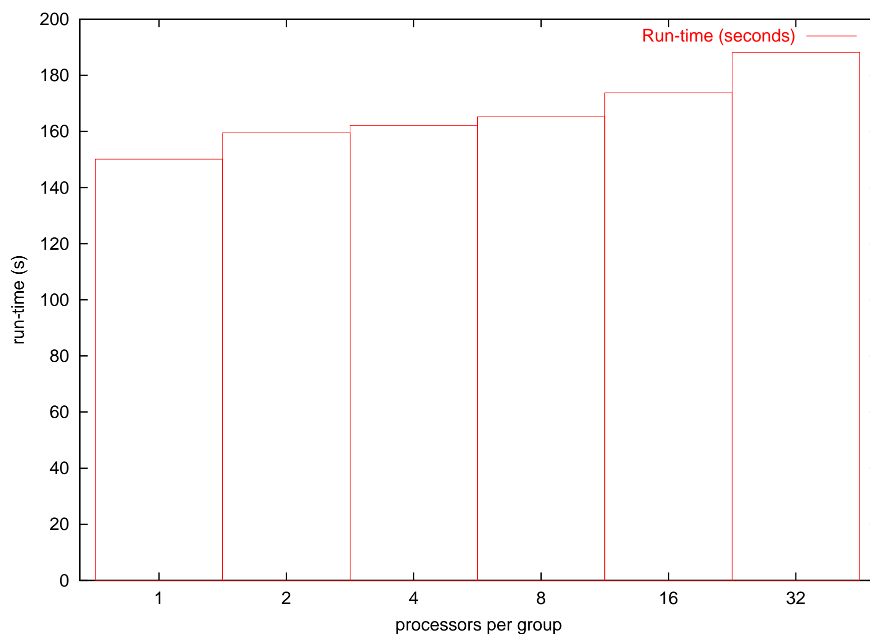

Fig. 8.15. Run-time of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the 249 760 state Voting model for 165 s-points on the Viking Beowulf cluster.

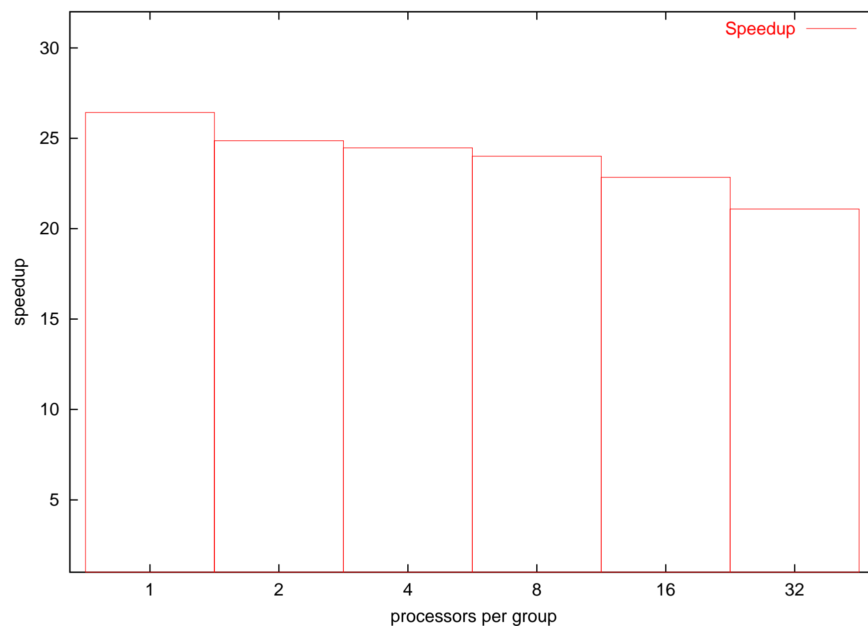

Fig. 8.16. Speedup of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the 249 760 state Voting model for 165 s-points.

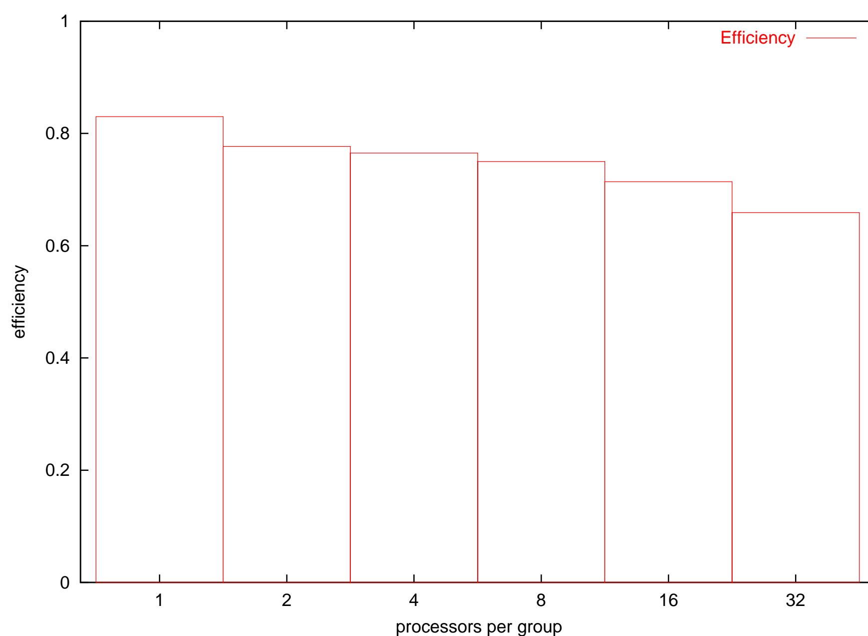

Fig. 8.17. Efficiency of hypergraph-partitioned sparse matrix–vector multiplication when using 32 processors in groups of varying sizes. Calculated for the 249 760 state Voting model for 165 s-points on the Viking Beowulf cluster.

Table 8.5 and Figs. 8.15, 8.16 and 8.17 show the hypergraph scalability in the case where 32 slave processors were divided into various size sub-clusters (32 groups of 1, 16 groups of 2, 8 groups of 4, and so on). This was to measure the benefit to be gained from adding extra groups to draw s-points from the global work queue versus doing the computation across larger groups of slave processors (which may be necessary when the state space of the model under analysis is very large). The efficiency decreases as the number of groups decreases. Note, however, that with run-times of between 150 and 188 seconds, there is still a dramatic improvement over the run-time on a single slave processor ($3\,968$ seconds) regardless of the group size employed. These results suggest that, given a fixed number of slave processors, it is best to allocate them into the smallest size subgroups (that is, to maximise the number of groups drawing from the global work-queue) subject to the constraints imposed by the size of the model and the memory available on each processor.

Chapter 9

Conclusion

9.1 Summary of Achievements

This thesis has explored the numerical computation of passage time densities and quantiles in large Markov and semi-Markov models. Prior work in this area has focused mainly on the analysis of Markov systems using the two techniques described in Chapter 3, viz. the Laplace transform method and uniformization. Previous attempts to analyse semi-Markov models have, however, been stymied by the complexity of representing the generally-distributed state sojourn-time functions in a space-efficient manner. This thesis rectifies this by presenting techniques which enable the computation of passage time densities and quantiles in Markov and semi-Markov models with state-spaces of 10^7 states and above.

The key contribution of this thesis is the iterative passage time analysis algorithm described in Chapter 4. This calculates the Laplace transform of the passage time quantity by convolving the Laplace transforms of the state holding-times across all possible paths between source and target states. This Laplace transform is then inverted using numerical inversion techniques to yield the value of the density or quantile function at a range of user-specified time points. This iterative algorithm has also been extended to permit the efficient calculation of transient state distributions.

It is possible to adopt an efficient representation scheme for the state holding-time

functions because of the way in which the numerical Laplace transform inversion algorithms work. In order to calculate the value of a function $f(t)$ at a range of t-points, both the Euler and the Laguerre methods require the value of the corresponding Laplace transform $f^*(s)$ at a range of s-points, and these s-points can be determined in advance and do not change during inversion. Rather than attempt to maintain full symbolic representations of the state holding-time functions and of their convolutions, therefore, we instead only store the values of the functions at these s-points. This requires constant storage space regardless of the number of convolutions performed.

Two techniques were investigated to deal with the high memory consumption and long run-times experienced when analysing very large models. The first approach was to perform passage time analysis in parallel. This takes advantage of the distributed memory of a cluster of workstations or parallel computer to store the very large transition matrices. Furthermore, parallel computation offers the potential to perform this analysis faster than would be possible on a single machine. Central to the efficiency of this approach is the way in which non-zero elements of the matrix are assigned to the processors involved. As communication is very expensive in parallel computation (compared to local computation), it is vital that these non-zeros be assigned such that the communication required between processors at the beginning of each iteration is as small as possible. This is achieved by applying hypergraph partitioning to the matrix. Experimental comparison with simple row-striped partitioning shows that hypergraph-partitioned sparse matrix–vector multiplication is faster and scales better.

The second approach explored was to reduce the size of the model's state-space (and hence the dimensions of its transition matrix) by aggregating states together in such a way that sojourn-time distribution information is preserved. An algorithm for semi-Markov processes which achieves this was implemented and then evaluated on a variety of different size models. We observed that a tension exists between attempting to minimise the fill-in of the matrix (as states are aggregated, new transitions are created) and trying to keep low the number of aggregation operations (convolution, branching and cycle removals) which have to be performed. Aggregating states results in fill-in of the matrix and consequently makes it more dense, but attempting to avoid or delay

this results in an increase in the number of aggregation operations performed. The order in which states are selected for aggregation has a significant impact on the behaviour of the algorithm – in particular, choosing states with the smallest number of paths through them delays the observed explosion in the number of aggregation operations the longest while keeping matrix density low. We also investigated the feasibility of performing aggregation in parallel. We noted that it is important to ensure that the number of non-zeros in the final aggregated matrix does not exceed the number in the original matrix, even if the dimensions of the aggregated matrix are much smaller than the original matrix.

In order to describe large semi-Markov models succinctly, we have devised a high-level modelling formalism called Semi-Markov Stochastic Petri Nets (SM-SPNs). This is an extension of stochastic Petri nets which includes transitions with generally-distributed firing delays. In the event that two or more general transitions are concurrently enabled, the selection of the next to fire is done by probabilistic choice and the delay is then sampled from that transition's firing distribution. This means that the underlying stochastic process is isomorphic to a semi-Markov chain. While SM-SPNs do not attempt to model true Generalised Semi-Markov Process-style concurrency, they do support Markovian concurrency provided that generally-distributed (non-exponential) transitions are exclusively enabled.

We have also presented a language for the framing of performance queries about semi-Markov models in a rigorous manner. This extended Continuous Stochastic Logic (eCSL) permits the construction of steady-state, transient and passage time questions at the SM-SPN level (i.e. in terms of the markings of the places in an SM-SPN) rather than in terms of the states of the underlying SMP. We believe this is a more natural mode of expression for performance modellers.

We have drawn on the work described above and on the DNAmaca steady-state Markov chain analyser to implement three analysis tools. HYDRA (HYpergraph-based Distributed Response Time Analyser) uses uniformization and hypergraph partitioning to analyse very large Markov models for transient and passage time quantities. SMCA (Semi-Markov Chain Analyser) is a steady-state analyser for large semi-Markov mod-

els. Finally, SMARTA (Semi-MArkov Response Time Analyser) uses the iterative algorithm and numerical Laplace transform inversion to compute the passage time measure of interest in very large semi-Markov models. Using these tools, passage time density and quantile results have been calculated for Markov and semi-Markov models with up to 15.4 million states. The HYDRA and SMARTA are shown to scale well, especially on architectures with fast interconnection networks.

9.2 Applications

In this section we discuss the applicability of the contributions of this thesis to the area of performance analysis.

As described in the introduction to this thesis, response time targets can be found in many areas. It would be interesting to apply the tools and techniques presented here to models other than those of computer and communication systems. One such area could be the modelling of accident and emergency units in NHS hospitals. These departments aim to see 90% of patients within 4 hours of admission and at present collect a large amount of data about patient waiting times. With access to this data it could be possible to model and then investigate their performance using the techniques described here. This would allow predictions of resource utilisation and patient waiting times to be made, which could then be used to target bottlenecks and hence improve response times. It would also allow the potential impact of different resource allocation decisions (including extra staff or equipment) to be investigated. Indeed, such analysis need not be restricted solely to A&E departments, but could also applied to other areas such as ambulance services and out-patient clinics.

The examples presented in this thesis indicate that numerical Laplace transform inversion is a viable technique for recovering $f(t)$ from $f^*(s)$ for a broad range of probability distributions. This could be of use in other areas of performance analysis where analytical results are expressed in terms of Laplace transforms which cannot be symbolically inverted (e.g. the sojourn time distribution in an MM CPP/GE/c G-Queue [69]).

Our investigations have demonstrated that hypergraph partitioning is an efficient matrix partitioning scheme for sparse matrix–vector multiplication. It is used here in parallel computation of response time densities, but it could also be applied to parallel steady-state solution of very large Markov chains. A number of such parallel tools already exist, but to the best of our knowledge do not make use of partitioning schemes which minimise communication. It would be interesting to see what the effect of its adoption would be on their run-times and scalability.

9.3 Future Work

There are a number of possible extensions to the work presented in this thesis. By implementing a fully parallel pipeline with parallel state-space generation, steady-state solution and hypergraph partitioning, the size of models which can analysed could be increased further, perhaps even up to 10^8 states. It is worth noting, however, that even on the fastest hardware currently available the time taken to perform the passage time analysis on such models would be prohibitive. For the 15.4 million state semi-Markov model analysed in this thesis, it took nearly three days to compute the passage time densities on 64 2GHz Intel Xeon machines using SMARTA.

Another area of future research could be the graphical specification and automatic translation of eCSL queries into the modified DNAmaca input language. At present, the user must manually convert eCSL formulae into passage time specifications, with the possibility for error which that entails.

The parallel aggregation algorithm merits further investigation. From the results presented in Table 5.5, the algorithm offers the ability to aggregate a large proportion of a state-space without the need for interprocessor communication. More research is needed, however, on determining how much of the state-space can be aggregated without introducing so many new non-zeros that the aggregated matrix actually contains more than the original. Also, the cost of aggregating and solving the aggregated matrix needs to be compared with solving the original matrix.

It would be interesting to investigate a compositional approach to passage time analy-

sis. This would be well suited to stochastic process algebra models, and would involve computing passage time densities in subcomponents which would then be combined to yield an overall passage time density. Such an approach would have the benefit that the passage time analysis of the individual components should be faster than analysis of the whole system as the sum of state-spaces of the individual components will usually be much smaller than the total state-space of the entire model. Research would need to be undertaken, however, into exactly how the results from components should be combined to give system-level results, and how good an approximation can be achieved.

Finally, the semi-Markov chains analysed in this thesis do not model true concurrency in the case where multiple concurrently-enabled general transitions exist. While this is acceptable in certain situations, it can only be an approximation to genuine concurrency. Some formalisms (such as Deterministic and Stochastic Petri Nets [96]) do model true concurrency in the presence of concurrent non-exponential (although not necessarily fully general) transitions but the computations required even to calculate the steady-state probabilities are complex. An obvious extension of the work presented in this thesis would therefore be the computation of passage time densities and quantiles in true-concurrency models with multiple concurrently-enabled non-exponential transitions.

Appendix A

Models

This appendix gives full details of the five high-level models which are referred to in the main text. These are a GSPN model of communications protocol, a GSPN model of a manufacturing system, a tree-like queueing network, an SM-SPN model of an electronic voting system and an SM-SPN model of a parallel web-server. In each case, we provide the full specification of the model in the language described in Chapter 6.

A.1 Courier Communications Protocol

The GSPN shown in Fig. A.1 (originally presented in [125]) models the ISO Application, Session and Transport layers of the Courier sliding-window communication protocol. Data flows from a sender (p_1 to p_{26}) to a receiver (p_{27} to p_{46}) via a network. The sender's transport layer fragments outgoing data packets; this is modelled as two paths between p_{13} and p_{35}. The path via t_8 carries all fragments before the last one through the network to p_{33}. Acknowledgements for these fragments are sent back to the sender (as signalled by the arrival of a token on p_{20}), but no data is delivered to the higher layers on the receiver side. The path via t_9 carries the last fragment of each message block. Acknowledgements for these fragments are generated and a data token is delivered to higher receiver layers via t_{27}.

The average number of data packets sent is determined by the ratio of the weights on

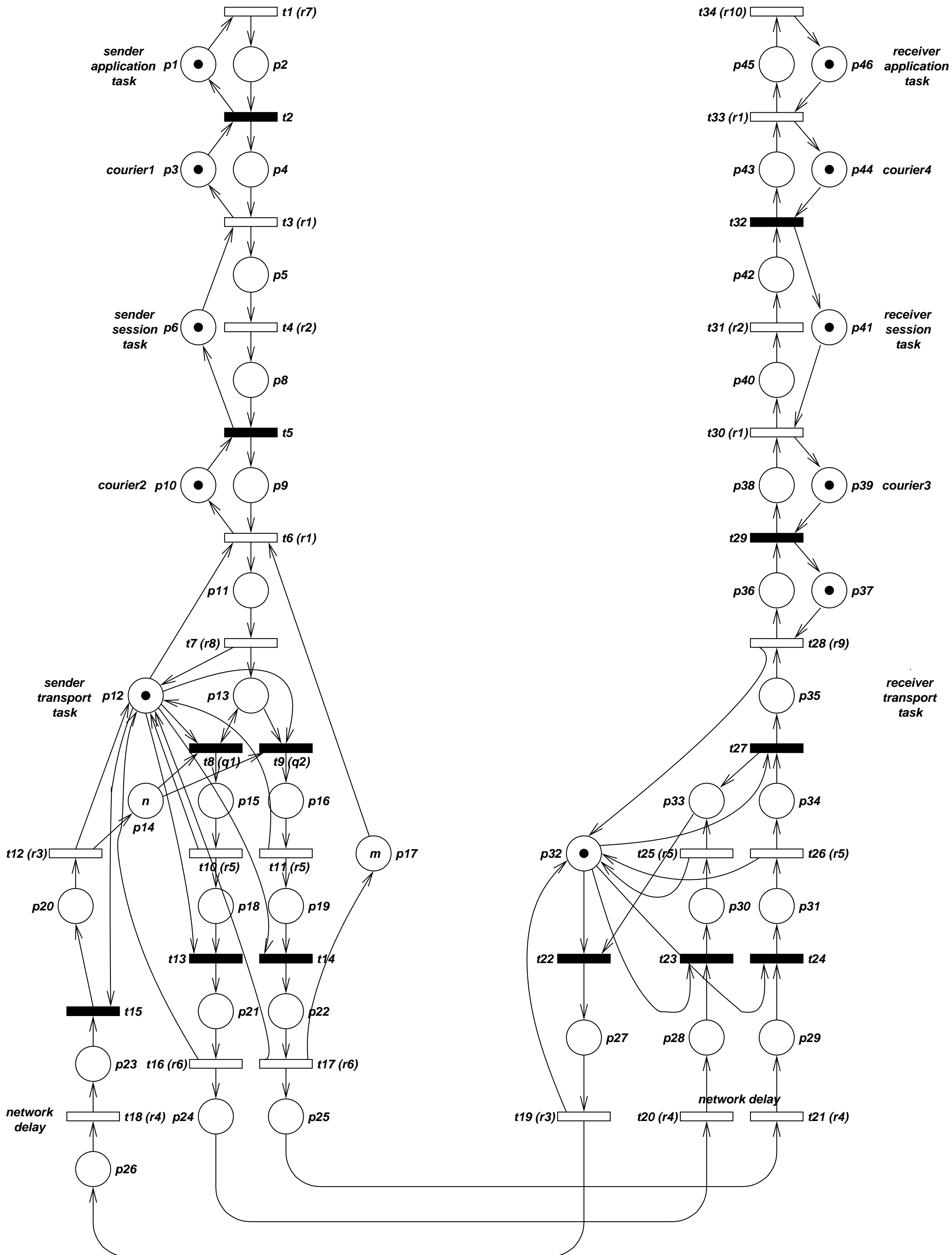

Fig. A.1. The Courier communications protocol GSPN model [125].

the immediate transitions t_8 and t_9. This ratio, known as the fragmentation ratio, is given by $q_1 : q_2$ (where q_1 and q_2 are the weights associated with transitions t_8 and t_9 respectively). This number of data packets is geometrically distributed, with parameter $q_1/(q_1 + q_2)$. In our case study, we use a fragmentation ratio of one.

The transport layer is further characterised by two important parameters: the sliding window size n (p_{14}) and the transport space m (p_{17}). For our example, we set $m = 1$ and $n = 1$.The transition rates $r1, r2, \ldots, r10$ used in the original model [125] were obtained by benchmarking a working implementation of the protocol.

```
\model{

  \constant{nn}{1}
  \constant{mm}{1}

  \constant{r1}{(5000.0/0.57)}
  \constant{r2}{(5000.0/4.97)}
  \constant{r3}{(5000.0/1.09)}
  \constant{r4}{(5000.0/10.37)}
  \constant{r5}{(5000.0/4.29)}
  \constant{r6}{(5000.0/0.39)}
  \constant{r7}{(5000.0/0.68)}
  \constant{r8}{(5000.0/2.88)}
  \constant{r9}{(5000.0/3.45)}
  \constant{r10}{(5000.0/1.25)}

  \constant{q1}{1.0}
  \constant{q2}{1.0}

  \statevector{
    \type{short}{  p1,  p2,  p3,  p4,  p5,  p6,  p8,  p9, p10 }
    \type{short}{ p11, p12, p13, p14, p15, p16, p17, p18, p19 }
    \type{short}{ p20, p21, p22, p23, p24, p25, p26, p27, p28 }
    \type{short}{ p29, p30, p31, p32, p33, p34, p35, p36, p37 }
    \type{short}{ p38, p39, p40, p41, p42, p43, p44, p45, p46 }
  }

  \initial{
    p1 =  p3 =  p6 = p10 = p12 = p32 = 1;
    p37 = p39 = p41 = p44 = p46 = 1;
    p2 =  p4 =  p5 =  p8 = p9  = p11 = 0;
    p13 = p15 = p16 = p18 = p19 = 0;
    p20 = p21 = p22 = p23 = p24 = p25 = 0;
    p26 = p27 = p28 = p29 = 0;
    p30 = p31 = p33 = p34 = p35 = p36 = 0;
    p38 = p40 = p42 = p43 = p45 = 0;
    p14 = nn; p17 = mm;
  }

  \transition{t1}{
```

```
    \condition{p1 >  0}
    \action{next->p1 = p1 - 1; next->p2 = p2 + 1;}
    \rate{r7}
  }

  \transition{t2}{
    \condition{p2 > 0 && p3 > 0}
    \action{next->p2 = p2 - 1; next->p3 = p3 - 1;
            next->p1 = p1 + 1; next->p4 = p4 + 1;}
    \weight{1.0}
  }

  \transition{t3}{
    \condition{p4 > 0 && p6 > 0}
    \action{next->p4 = p4 - 1; next->p6 = p6 - 1;
            next->p3 = p3 + 1; next->p5 = p5 + 1;}
    \rate{r1}
  }

  \transition{t4}{
    \condition{p5 > 0}
    \action{next->p5 = p5 - 1; next->p8 = p8 + 1;}
    \rate{r2}
 }

  \transition{t5}{
    \condition{p8 > 0 && p10 > 0}
    \action{next->p8 = p8 - 1; next->p10 = p10 - 1;
            next->p6 = p6 + 1; next->p9 = p9 + 1;}
    \weight{1.0}
  }

  \transition{t6}{
    \condition{p9 > 0 && p12 > 0 && p17 > 0}
    \action{next->p9 = p9 - 1; next->p12 = p12 - 1;
            next->p17 = p17 - 1; next->p10 = p10 + 1;
            next->p11 = p11 + 1;}
    \rate{r1}
  }

  \transition{t7}{
    \condition{p11 > 0}
    \action{next->p11 = p11 - 1; next->p12 = p12 + 1;
            next->p13 = p13 + 1;}
    \rate{r8}
  }

  \transition{t8}{
    \condition{p12 > 0 && p13 > 0 && p14 > 0}
    \action{next->p12 = p12 - 1; next->p14 = p14 - 1;
            next->p15 = p15 + 1;}
    \weight{q1}
  }

  \transition{t9}{
```

```
    \condition{p12 > 0 && p13 > 0 && p14 > 0}
    \action{next->p12 = p12 - 1; next->p13 = p13 - 1;
            next->p14 = p14 - 1; next->p16 = p16 + 1;}
    \weight{q2}
  }

  \transition{t10}{
    \condition{p15 > 0}
    \action{next->p15 = p15 - 1; next->p12 = p12 + 1;
            next->p18 = p18 + 1;}
    \rate{r5}
  }

  \transition{t11}{
    \condition{p16 > 0}
    \action{next->p16 = p16 - 1; next->p12 = p12 + 1;
            next->p19 = p19 + 1;}
    \rate{r5}
  }

  \transition{t12}{
    \condition{p20 > 0}
    \action{next->p20 = p20 - 1; next->p14 = p14 + 1;
            next->p12 = p12 + 1;}
    \rate{r3}
  }

  \transition{t13}{
    \condition{p12 > 0 && p18 > 0}
    \action{next->p12 = p12 - 1; next->p18 = p18 - 1;
            next->p21 = p21 + 1;}
    \weight{1.0}
  }

  \transition{t14}{
    \condition{p12 > 0 && p19 > 0}
    \action{next->p12 = p12 - 1; next->p19 = p19 - 1;
            next->p22 = p22 + 1;}
    \weight{1.0}
  }

  \transition{t15}{
    \condition{p12 > 0 && p23 > 0}
    \action{next->p12 = p12 - 1; next->p23 = p23 - 1;
            next->p20 = p20 + 1;}
    \weight{1.0}
  }

  \transition{t16}{
    \condition{p21 > 0}
    \action{next->p21 = p21 - 1; next->p12 = p12 + 1;
            next->p24 = p24 + 1;}
    \rate{r6}
  }
```

```
\transition{t17}{
  \condition{p22 > 0}
  \action{next->p22 = p22 - 1; next->p12 = p12 + 1;
          next->p17 = p17 + 1; next->p25 = p25 + 1;}
  \rate{r6}
}

\transition{t18}{
  \condition{p26 > 0}
  \action{next->p26 = p26 - 1; next->p23 = p23 + 1;}
  \rate{r4}
}

\transition{t19}{
  \condition{p27 > 0}
  \action{next->p27 = p27 - 1; next->p32 = p32 + 1;
          next->p26 = p26 + 1;}
  \rate{r3}
}

\transition{t20}{
  \condition{p24 > 0}
  \action{next->p24 = p24 - 1; next->p28 = p28 + 1;}
  \rate{r4}
}

\transition{t21}{
  \condition{p25 > 0}
  \action{next->p25 = p25 - 1; next->p29 = p29 + 1;}
  \rate{r4}
}

\transition{t22}{
  \condition{p32 > 0 && p33 > 0}
  \action{next->p32 = p32 - 1; next->p33 = p33 - 1;
          next->p27 = p27 + 1;}
  \weight{1.0}
}

\transition{t23}{
  \condition{p32 > 0 && p28 > 0}
  \action{next->p32 = p32 - 1; next->p28 = p28 - 1;
          next->p30 = p30 + 1;}
  \weight{1.0}
}

\transition{t24}{
  \condition{p32 > 0 && p29 > 0}
  \action{next->p32 = p32 - 1; next->p29 = p29 - 1;
          next->p31 = p31 + 1;}
  \weight{1.0}
}

\transition{t25}{
  \condition{p30 > 0}
```

```
    \action{next->p30 = p30 - 1; next->p32 = p32 + 1;
            next->p33 = p33 + 1;}
    \rate{r5}
  }

  \transition{t26}{
    \condition{p31 > 0}
    \action{next->p31 = p31 - 1; next->p34 = p34 + 1;
            next->p32 = p32 + 1;}
    \rate{r5}
  }

  \transition{t27}{
    \condition{p34 > 0 && p32 > 0}
    \action{next->p34 = p34 - 1; next->p32 = p32 - 1;
            next->p33 = p33 + 1; next->p35 = p35 + 1;}
    \weight{1.0}
  }

  \transition{t28}{
    \condition{p35 > 0 && p37 > 0}
    \action{next->p35 = p35 - 1; next->p37 = p37 - 1;
            next->p32 = p32 + 1; next->p36 = p36 + 1;}
    \rate{r9}
  }

  \transition{t29}{
    \condition{p36 > 0 && p39 > 0}
    \action{next->p36 = p36 - 1; next->p39 = p39 - 1;
            next->p38 = p38 + 1; next->p37 = p37 + 1;}
    \weight{1.0}
  }

  \transition{t30}{
    \condition{p41 > 0 && p38 > 0}
    \action{next->p41 = p41 - 1; next->p38 = p38 - 1;
            next->p39 = p39 + 1; next->p40 = p40 + 1;}
    \rate{r1}
  }

  \transition{t31}{
    \condition{p40 > 0}
    \action{next->p40 = p40 - 1; next->p42 = p42 + 1;}
    \rate{r2}
  }

  \transition{t32}{
    \condition{p42 > 0 && p44 > 0}
    \action{next->p42 = p42 - 1; next->p44 = p44 - 1;
            next->p43 = p43 + 1; next->p41 = p41 + 1;}
    \weight{1.0}
  }

  \transition{t33}{
    \condition{p43 > 0 && p46 > 0}
```

```
    \action{next->p43 = p43 - 1; next->p46 = p46 - 1;
            next->p44 = p44 + 1; next->p45 = p45 + 1;}
    \rate{r1}
  }

  \transition{t34}{
    \condition{p45 > 0}
    \action{next->p45 = p45 - 1; next->p46 = p46 + 1;}
    \rate{r10}
  }
}
```

A.2 Flexible Manufacturing System

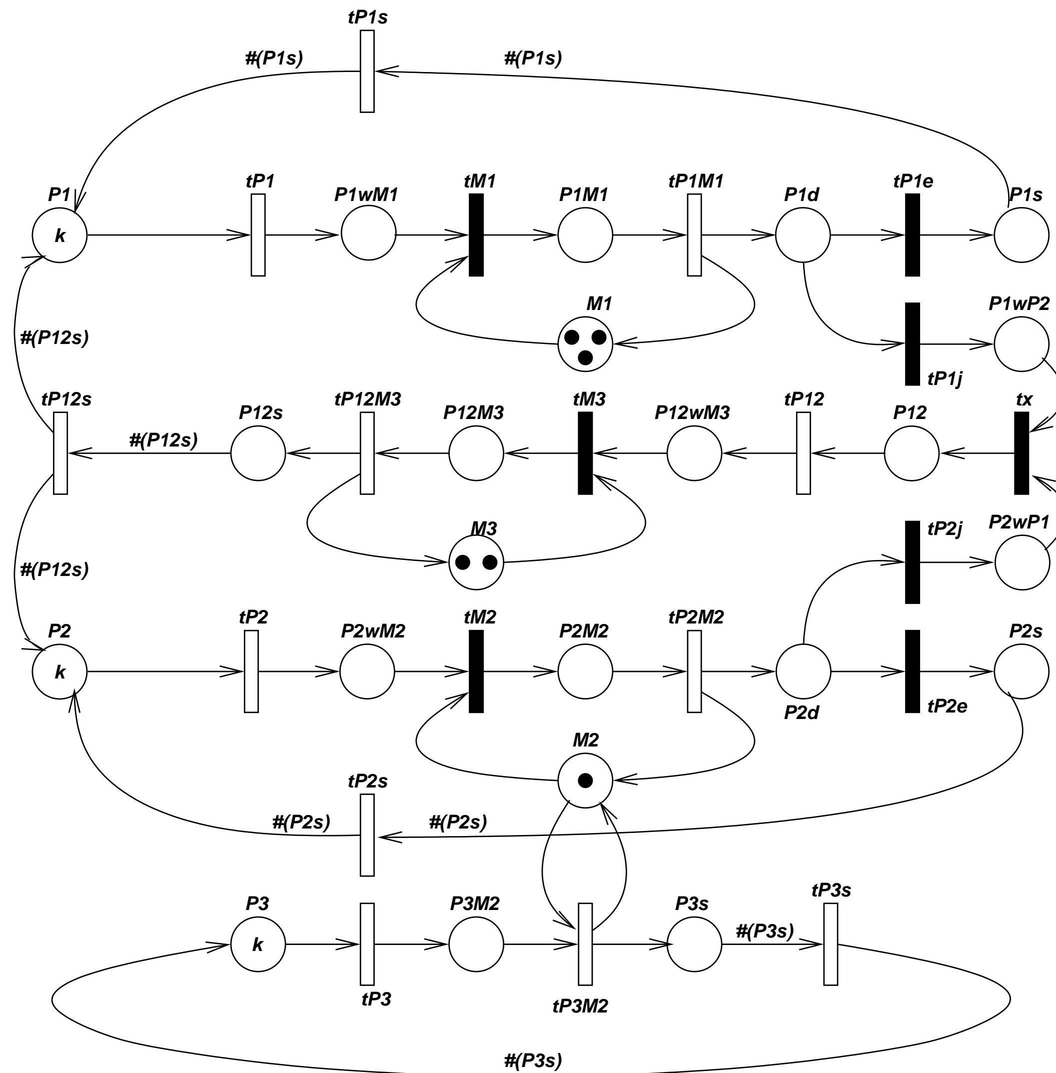

Fig. A.2. The GSPN model of a Flexible Manufacturing System [41].

Fig. A.2 shows a 22-place GSPN model of a flexible manufacturing system [41]. The model describes an assembly line with three types of machines ($M1$, $M2$ and $M3$) which assemble four types of parts ($P1$, $P2$, $P3$ and $P12$). Initially, there are k unprocessed parts of each type $P1$, $P2$ and $P3$ in the system. There are no parts of type $P12$ at start-up since these are assembled from processed parts of type $P1$ and $P2$ by the machines of type $M3$. When parts of any type are finished, they are stored for shipping on places $P1s$, $P2s$, $P3s$ and $P12s$.

```
\model{

  \constant{kk}{7}

  \statevector{
    \type{short}{P1, P1wM1, P1M1, P1d, P1s, M1, P1wP2}
    \type{short}{P12s, P12M3, M3, P12wM3, P12}
    \type{short}{P2, P2wM2, P2M2, M2, P2d, P2s, P2wP1}
    \type{short}{P3, P3M2, P3s}
  }

  \initial{
    P1 = kk; P1wM1 = 0; P1M1 = 0; P1d = 0; P1s = 0; M1 = 3;
    P1wP2 = 0; P12s = 0; P12M3 = 0; M3 = 2; P12wM3 = 0;
    P12 = 0; P2 = kk; P2wM2 = 0; P2M2 = 0; M2 = 1; P2d = 0;
    P2s = 0; P2wP1 = 0; P3 = kk; P3M2 = 0; P3s = 0;
  }

  \transition{tP1s}{
    \condition{P1s > 0}
    \action{next->P1s = 0; next->P1 += P1s;}
    \rate{1.0}
  }

  \transition{tP1}{
    \condition{P1 > 0}
    \action{next->P1 = P1 - 1; next->P1wM1 = P1wM1 + 1;}
    \rate{1.0}
  }

  \transition{tM1}{
    \condition{(P1wM1 > 0) && (M1 > 0)}
    \action{next->P1wM1 = P1wM1 - 1; next->M1 = M1 - 1;
            next->P1M1 = P1M1 + 1;}
    \weight{1.0}
  }

  \transition{tP1M1}{
    \condition{P1M1 > 0}
    \action{next->P1M1 = P1M1 - 1; next->M1 = M1 + 1;
            next->P1d = P1d + 1;}
    \rate{1.0}
```

```
}

\transition{tP1e}{
  \condition{P1d > 0}
  \action{next->P1d = P1d - 1; next->P1s = next->P1s + 1;}
  \weight{1.0}
}

\transition{tP1j}{
  \condition{P1d > 0}
  \action{next->P1d = P1d - 1; next->P1wP2 = next->P1wP2 + 1;}
  \weight{1.0}
}

\transition{tP12s}{
  \condition{P12s > 0}
  \action{next->P12s = 0; next->P1 = P1 + P12s;
          next->P2 = P2 + P12s;}
  \rate{1.0}
}

\transition{tP12M3}{
  \condition{P12M3 > 0}
  \action{next->P12M3 = P12M3 - 1; next->P12s = P12s + 1;
     next->M3 = M3 + 1;}
  \rate{1.0}
}

\transition{tM3}{
  \condition{(M3 > 0) && (P12wM3 > 0)}
  \action{next->P12wM3 = P12wM3 - 1; next->M3 = M3 - 1;
          next->P12M3 = P12M3 + 1;}
  \weight{1.0}
}

\transition{tP12}{
  \condition{P12 > 0}
  \action{next->P12 = P12 - 1; next->P12wM3 = P12wM3 + 1;}
  \rate{1.0}
}

\transition{tx}{
  \condition{(P1wP2 > 0) && (P2wP1 > 0)}
  \action{next->P1wP2 = P1wP2 - 1; next->P2wP1 = P2wP1 - 1;
          next->P12 = P12 + 1;}
  \weight{1.0}
}

\transition{tP2}{
  \condition{P2 > 0}
  \action{next->P2 = P2 - 1; next->P2wM2 = P2wM2 + 1;}
  \rate{1.0}
}

\transition{tM2}{
```

```
    \condition{(P2wM2 > 0) && (M2 > 0)}
    \action{next->P2wM2 = P2wM2 - 1; next->M2 = M2 - 1;
            next->P2M2 = P2M2 + 1;}
    \weight{1.0}
  }

  \transition{tP2M2}{
    \condition{P2M2 > 0}
    \action{next->P2M2 = P2M2 - 1; next->P2d = P2d + 1;
            next->M2 = M2 + 1;}
    \rate{1.0}
  }

  \transition{tP2j}{
    \condition{P2d > 0}
    \action{next->P2d = P2d - 1; next->P2wP1 = P2wP1 + 1;}
    \weight{1.0}
  }

  \transition{tP2e}{
    \condition{P2d > 0}
    \action{next->P2d = P2d - 1; next->P2s = P2s + 1;}
    \weight{1.0}
  }

  \transition{t2Ps}{
    \condition{P2s > 0}
    \action{next->P2s = 0; next->P2 = P2 + P2s; }
    \rate{1.0}
  }

  \transition{tP3}{
    \condition{P3 > 0}
    \action{next->P3 = P3 - 1; next->P3M2 = P3M2 + 1;}
    \rate{1.0}
  }

  \transition{tP3M2}{
    \condition{(M2 > 0) && (P3M2 > 0)}
    \action{next->P3M2 = P3M2 - 1; next->P3s = P3s + 1;}
    \rate{1.0}
  }

  \transition{tP3s}{
    \condition{P3s > 0}
    \action{next->P3s = 0; next->P3 = P3 + P3s;}
    \rate{1.0}
  }
}
```

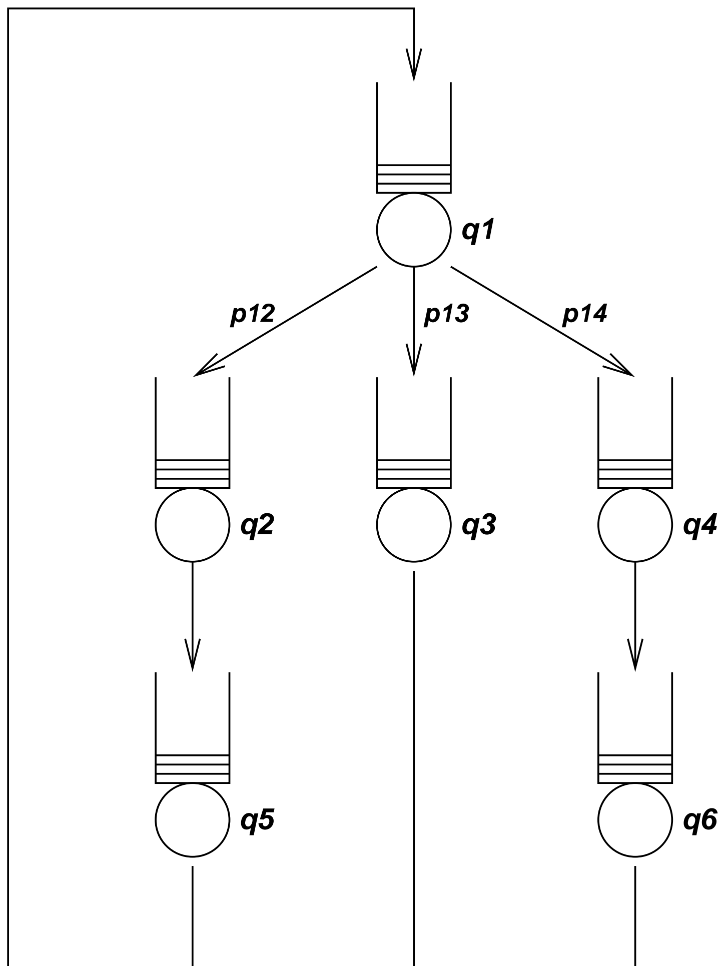

Fig. A.3. The tree-like queueing network [68, 70].

A.3 Tree-like Queueing Network

Fig. A.3 shows a tree-like queueing network which has six servers with rates $\mu_1, \ldots, \mu_6$ and non-zero routing probabilities as shown. Thus the visitation rates $v_1, \ldots, v_6$ for servers 1 to 6 are respectively proportional to: $1,\ \ p_{12},\ \ p_{13},\ \ p_{14},\ \ p_{12},\ \ p_{14}$. For this example, we set $\{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6\} = \{3, 5, 4, 6, 2, 1\}$ and $\{p_{12}, p_{13}, p_{14}\} = \{0.2, 0.5, 0.3\}$.

Analytical results for the cycle time density in this type of overtake-free, tree-like queueing network with M servers and population n are known [68, 70]. In particular, if the servers in an overtake-free path $(1, 2, ..., m)$ $(m \leq M)$ have distinct service rates $\mu_1, \mu_2, ..., \mu_m$, the passage time density function, conditional on the choice of path, is

$$\frac{\prod_{i=1}^{m} \mu_i}{G(n-1)} \sum_{c=0}^{n-1} G_m(n-c-1) \sum_{j=1}^{m} \frac{e^{-\mu_j t}}{\prod_{1 \leq i \neq j \leq m} (\mu_i - \mu_j)} \sum_{i=0}^{c} \frac{(v_j t)^{c-i}}{(c-i)!} K^m(j, i)$$

where $K^m(j, l)$, $G_m(n-c-1)$ and $G(n-1)$ are normalising constants that may be computed efficiently by Buzen's algorithm [33]. If we define the recursive function k, for real vector $\mathbf{y} = (y_1, \ldots, y_a)$ and integers a, b $(0 \leq a \leq M,\ \ 0 \leq b \leq N-1)$ by:

$$\begin{aligned}
k(\mathbf{y}, a, b) &= k(\mathbf{y}, a-1, b) + y_a k(\mathbf{y}, a, b-1) \quad (a, b > 0) \\
k(\mathbf{y}, a, 0) &= 1 \quad (a > 0) \\
k(\mathbf{y}, 0, b) &= 0 \quad (b \geq 0)
\end{aligned}$$

then:

$$\begin{aligned}
G_m(l) &= k(\mathbf{x}_m, M-m, l) \quad (0 \leq l \leq n-1) \\
G(n-1) &= k(\mathbf{x}, M, n-1) \\
K^m(j, l) &= k(\mathbf{w}_j, m-1, l)
\end{aligned}$$

with $x_i = v_i/\mu_i$, $\mathbf{x} = (x_1, \ldots, x_M), \mathbf{x}_m = (x_{m+1}, \ldots, x_M)$ and, for $1 \leq j \leq m$,

$$(\mathbf{w}_j)_k = \begin{cases} (v_k - v_j)/(\mu_k - \mu_j) & \text{if } 1 \leq k < j \\ (v_{k+1} - v_j)/(\mu_{k+1} - \mu_j) & \text{if } j \leq k < m \end{cases}$$

To compute the cycle time density in this network in terms of its underlying Markov Chain using the uniformization technique described in this thesis requires the state

vector to be augmented by 4 extra components so that a "tagged" customer can be followed through the system. The extra components are: the queue containing the tagged customer m, the position of the tagged customer in that queue k (with $k \geq 0$), the cycle sequence number c (an alternating bit, flipped whenever the tagged customer joins $q1$) and a flag p indicating whether or not a passage has started.

```
\model{

  \constant{NN}{16}

  \constant{MM}{6}
  \constant{MU1}{3.0}
  \constant{MU2}{5.0}
  \constant{MU3}{4.0}
  \constant{MU4}{6.0}
  \constant{MU5}{2.0}
  \constant{MU6}{1.0}

  \constant{P1}{0.2}
  \constant{P2}{0.5}
  \constant{P3}{0.3}

  \statevector{\type{int}{n1, n2, n3, n4, n5, n6, m, k, c, p}}

  \initial{
     n1 = NN; n2 = n3 = n4 = n5 = n6 = 0; m = 1; k = NN - 1;
     c = 0; p = 0;
  }

  \transition{q1_q2_service}{
    \condition{n1 > 0}
    \action{
      next->n1 = n1 - 1;
      next->n2 = n2 + 1;
      next->p = 1;
      if (m==1) {
        if (k)
          next->k = k - 1;
        else {
          next->m = 2;
          next->k = n2;
        }
      }
    }
    \rate{MU1*P1/(P1+P2+P3)}
  }

  \transition{q1_q3_service}{
    \condition{n1 > 0}
    \action{
      next->n1 = n1 - 1;
      next->n3 = n3 + 1;
```

```
    next->p = 1;
    if (m==1) {
      if (k)
        next->k = k - 1;
      else {
        next->m = 3;
        next->k = n3;
      }
    }
  }
  \rate{MU1*P2/(P1+P2+P3)}
}

\transition{q1_q4_service}{
  \condition{n1 > 0}
  \action{
    next->n1 = n1 - 1;
    next->n4 = n4 + 1;
    next->p = 1;
    if (m==1) {
      if (k)
        next->k = k - 1;
      else {
        next->m = 4;
        next->k = n4;
      }
    }
  }
  \rate{MU1*P3/(P1+P2+P3)}
}

\transition{q2_q5_service}{
  \condition{n2 > 0}
  \action{
    next->n2 = n2 - 1;
    next->n5 = n5 + 1;
    next->p = 1;
    if (m==2) {
      if (k)
        next->k = k - 1;
      else {
        next->m = 5;
        next->k = n5;
      }
    }
  }
  \rate{MU2}
}

\transition{q3_q1_service}{
  \condition{n3 > 0}
  \action{
    next->n3 = n3 - 1;
    next->n1 = n1 + 1;
    if (m==3) {
```

```
        if (k)
          next->k = k - 1;
        else {
          next->m = 1;
          next->k = n1;
          next->c = !c;
          next->p = 0;
        }
      }
    }
    \rate{MU3}
  }

  \transition{q4_q6_service}{
    \condition{n4 > 0}
    \action{
      next->n4 = n4 - 1;
      next->n6 = n6 + 1;
      next->p = 1;
      if (m==4) {
        if (k)
          next->k = k - 1;
        else {
          next->m = 6;
          next->k = n6;
        }
      }
    }
    \rate{MU4}
  }

  \transition{q5_q1_service}{
    \condition{n5 > 0}
    \action{
      next->n5 = n5 - 1;
      next->n1 = n1 + 1;
      if (m==5) {
        if (k)
          next->k = k - 1;
        else {
          next->m = 1;
          next->k = n1;
          next->c = !c;
          next->p = 0;
        }
      }
    }
   \rate{MU5}
  }

  \transition{q6_q1_service}{
    \condition{n6 > 0}
    \action{
      next->n6 = n6 - 1;
      next->n1 = n1 + 1;
```

```
      if (m==6) {
        if (k)
          next->k = k - 1;
        else {
          next->m = 1;
          next->k = n1;
          next->c = !c;
          next->p = 0;
        }
      }
    }
    \rate{MU6}
  }
}
```

A.4 Voting Model

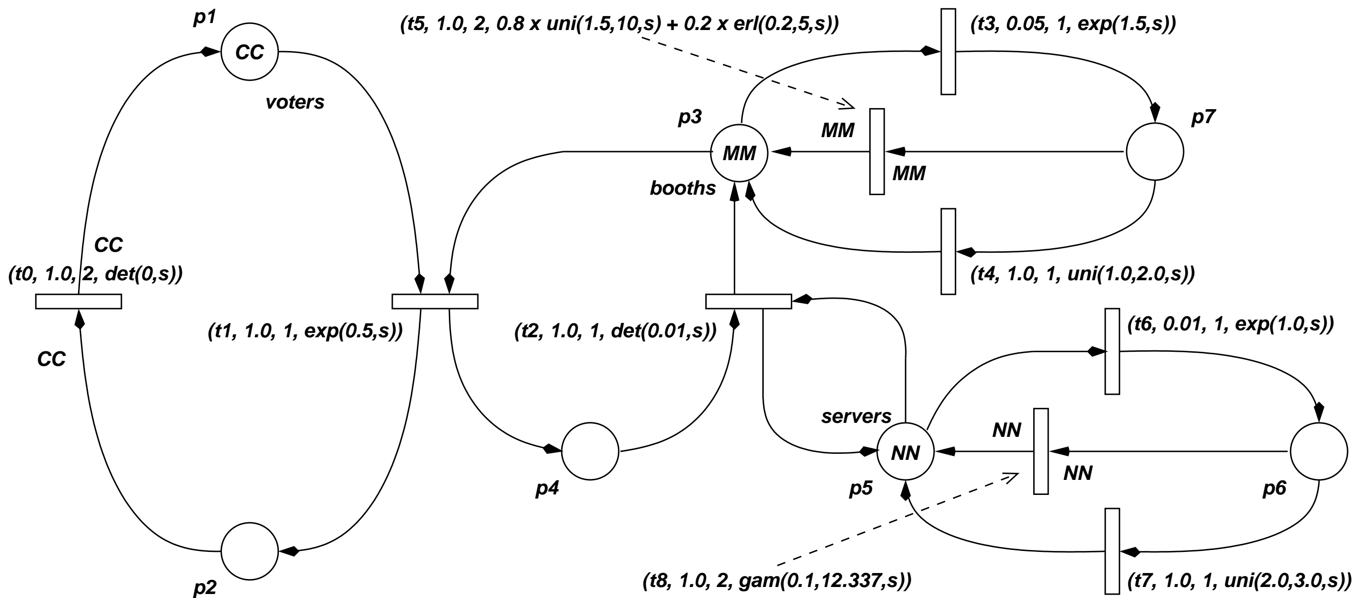

Fig. A.4. The Voting Model SM-SPN [24].

Fig. A.4 shows an SM-SPN model of a distributed voting system [24, 28, 29]. Voting agents vote asynchronously, moving from place p_1 to p_2 as they do so. A restricted number of polling units which receive their votes transit t_1 from place p_3 to place p_4. At t_2, the vote is registered with as many central voting units as are currently operational in p_5.

The system is considered to be in a failure mode if either all the polling units have

System	CC	MM	NN	States
1	11	7	4	2 081
2	22	7	4	4 050
3	60	25	4	106 540
4	100	30	4	249 760
5	125	40	4	541 280
6	150	40	5	778 850
7	175	45	5	1 140 050
8	300	80	10	10 991 400

Table A.1. Number of states generated by the Voting model SM-SPN in terms of the number of voters (CC), polling units (MM) and central voting units (NN).

failed and are in p_7 or all the central voting units have failed and are in p_6. If either of these complete failures occur, then a high priority repair is performed, which resets the failed units to a fully operational state. If some (but not all) the polling or voting units fail, they attempt self-recovery. The system will continue to function as long as at least one polling unit and one voting unit remain operational.

There are several voters, CC, a limited number of polling units, MM, and a smaller number of central voting units, NN. The size of the underlying semi-Markov chain can be varied by altering these three parameters as shown in Table A.1.

```
\model{

  %booths
  \constant{MM}{12}
  %servers
  \constant{NN}{4}
  %voters
  \constant{CC}{22}

  \statevector{
    \type{short}{ p1, p2, p3, p4, p5, p6, p7 }
  }

  \initial{
    p1 = CC; p2 = 0; p3 = MM; p4 = 0; p5 = NN; p6 = 0; p7 = 0;
  }
```

```
\transition{t1}{
  \condition{p1 > 0 && p3 > 0}
  \action{next->p1 = p1 - 1; next->p2 = p2 + 1;
          next->p3 = p3 - 1; next->p4 = p4 + 1;}
  \weight{1.0}
  \priority{1}
  \sojourntimeLT{ return exponential(0.5, s); }
}

\transition{t2}{
  \condition{p4 > 0 && p5 > 0}
  \action{next->p3 = p3 + 1; next->p4 = p4 - 1;}
  \weight{1.0}
  \priority{1}
  \sojourntimeLT{ return deterministic(0.01, s); }
}

\transition{t3}{
  \condition{p3 > 0}
  \action{next->p3 = p3 - 1; next->p7 = p7 + 1;}
  \weight{0.05}
  \priority{1}
  \sojourntimeLT{ return exponential(1.5, s); }
}

\transition{t4}{
  \condition{p7 > 0}
  \action{next->p3 = p3 + 1; next->p7 = p7 - 1;}
  \weight{1.0}
  \priority{1}
  \sojourntimeLT{ return uniform(1.0, 2.0, s); }
}

\transition{t5}{
  \condition{p7 > MM-1}
  \action{next->p3 = p3 + MM; next->p7 = p7 - MM;}
  \weight{1.0}
  \priority{2}
  \sojourntimeLT{ return (0.8 * uniform(1.5,10,s) +
      0.2 * erlang(0.2,5,s)); }
}

\transition{t6}{
  \condition{p5 > 0}
  \action{next->p5 = p5 - 1; next->p6 = p6 + 1;}
  \weight{0.01}
  \priority{1}
  \sojourntimeLT{ return exponential(1.0, s); }
}

\transition{t7}{
  \condition{p6 > 0}
  \action{next->p5 = p5 + 1; next->p6 = p6 - 1;}
  \weight{1.0}
  \priority{1}
```

```
    \sojourntimeLT{ return uniform(2.0, 3.0, s); }
  }

  \transition{t8}{
    \condition{p6 > NN-1}
    \action{next->p5 = p5 + NN; next->p6 = p6 - NN;}
    \weight{1.0}
    \priority{2}
    \sojourntimeLT{ return gamma(0.1, 12.337, s); }
  }

  \transition{t9}{
    \condition{p2 > CC-1}
    \action{next->p2 = p2 - CC; next->p1 = p1 + CC;}
    \weight{1.0}
    \priority{2}
    \sojourntimeLT{ return immediate(); }
  }
}
```

A.5 Web Content Authoring System

Fig. A.5 represents an SM-SPN model of a web server with RR clients (readers), WW web content authors (writers), SS parallel web servers and a write-buffer of BB in size [28, 29]. As with the Voting model, the size and complexity of the underlying semi-Markov chain can be varied by altering these four parameters as shown in Table A.2.

Clients can make read requests to one of the web servers for content (represented by the movement of tokens from p_8 to p_7). Web content authors submit page updates into the write buffer (represented by the movement of tokens from p_1 onto p_2 and p_4), and whenever there are no outstanding read requests all outstanding write requests in the buffer (represented by tokens on p_4) are applied to all functioning web servers (represented by tokens on p_6). Web servers can fail (represented by the movement of tokens from p_6 to p_5) and institute self-recovery unless all servers fail, in which case a high-priority recovery mode is initiated to restore all servers to a fully functional state. Complete reads and updates are represented by tokens on p_9 and p_2 respectively.

```
\model{

  % writers
```

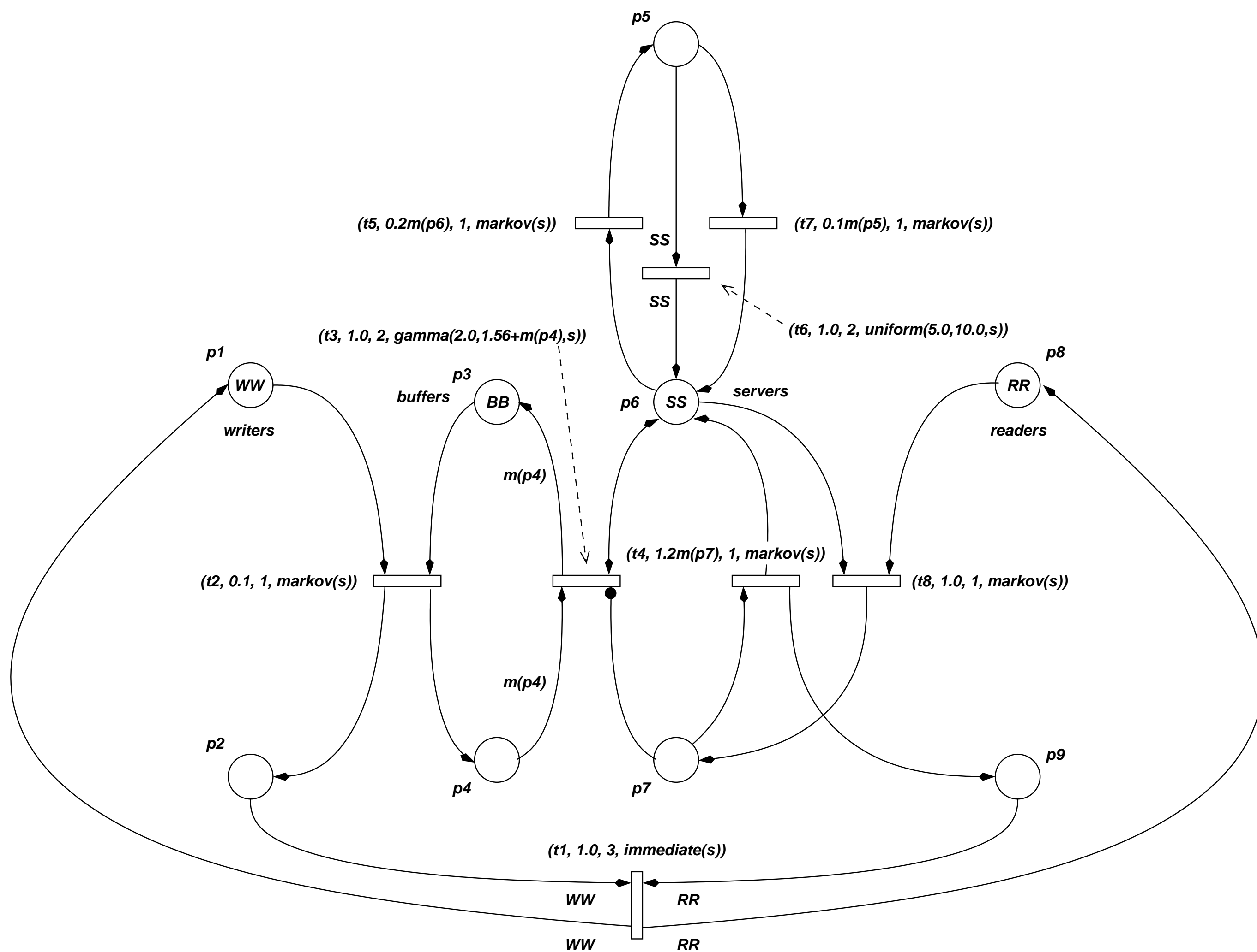

Fig. A.5. The Web-server Model SM-SPN [28, 29].

System	RR	WW	SS	BB	States
1	45	22	4	8	107 289
2	52	26	5	10	248 585
3	60	30	6	12	517 453
4	65	30	7	13	763 680
5	70	35	7	14	1 044 540
6	100	50	18	20	15 445 919

Table A.2. Number of states generated by the Web-server SM-SPN in terms of the number of clients (RR), authors (WW), parallel web servers (SS) and write-buffers (BB).

```
\constant{WW}{12}
% buffers
\constant{BB}{6}
% servers
\constant{SS}{3}
% readers
\constant{RR}{24}

% read-rate
\constant{RDRATE}{1.0}
% buffer-write-rate
\constant{BUFWRTRATE}{2.0}
% server-read-recovery
\constant{SRVRDRECOVER}{3.6}
% server-fail-rate
\constant{SRVFAILRATE}{0.2}
% server-fail-recovery
\constant{SRVFAILRECOVER}{(0.5*SRVFAILRATE)}

\statevector{
  \type{short}{ p1, p2, p3, p4, p5, p6, p7, p8, p9 }
}

\initial{
  p1 = WW; p2 = 0; p3 = BB; p4 = 0; p5 = 0;
  p6 = SS; p7 = 0; p8 = RR; p9 = 0;
}

\transition{t1}{
  \condition{ p2 == WW && p9 == RR}
  \action{next->p2 = p2 - WW; next->p1 = p1 + WW;
          next->p9 = p9 - RR; next->p8 = p8 + RR;}
  \weight{1.0}
  \priority{3}
  \sojourntimeLT{ return immediate(); }
}

\transition{t2}{
  \condition{ p1 > 0 && p3 > 0}
  \action{next->p3 = p3 - 1; next->p1 = p1 - 1;
          next->p2 = p2 + 1; next->p4 = p4 + 1;}
  \weight{ (0.1 * RDRATE) }
  \priority{1}
  \sojourntimeLT{ return markov(s); }
}

\transition{t3}{
  \condition{ p4 > 0 && p7 == 0 && p6 > 0 }
  \action{next->p3 = p3 + p4; next->p4 = 0;}
  \weight{1.0}
  \priority{2}
  \sojourntimeLT{ return gamma(BUFWRTRATE,(p4+1.56), s); }
}

\transition{t4}{
```

```
    \condition{ p7 > 0 }
    \action{next->p6 = p6 + 1;next->p7 = p7 - 1;
            next->p9 = p9 + 1;}
    \weight{p7*SRVRDRECOVER}
    \priority{1}
    \sojourntimeLT{ return markov(s); }
  }

  \transition{t5}{
    \condition{p6 > 0}
    \action{next->p5 = p5 + 1; next->p6 = p6 - 1;}
    \weight{p6*SRVFAILRATE}
    \priority{1}
    \sojourntimeLT{ return markov(s); }
  }

  \transition{t6}{
    \condition{ p5 == SS }
    \action{next->p5 = p5 - SS; next->p6 = p6 + SS;}
    \weight{1.0}
    \priority{2}
    \sojourntimeLT{ return uniform(5.0, 10.0, s); }
  }

  \transition{t7}{
    \condition{ p5 > 0 }
    \action{next->p5 = p5 - 1; next->p6 = p6 + 1;}
    \weight{p5*SRVFAILRECOVER}
    \priority{1}
    \sojourntimeLT{ return markov(s); }
  }

  \transition{t8}{
    \condition{ p6 > 0 && p8 > 0 }
    \action{next->p7 = p7 + 1; next->p6 = p6 - 1;
            next->p8 = p8 - 1;}
    \weight{RDRATE}
    \priority{1}
    \sojourntimeLT{ return markov(s); }
  }
}
```

Appendix B

Semi-Markov Process Aggregation Algorithm

$$
\begin{aligned}
aggregate_smp &: SMP \times S \rightarrow SMP \\
aggregate_smp & \quad (M, i) = fold(set, M', vs) \text{ where}
\end{aligned}
$$

$$
\begin{aligned}
vs &= aggregate_cycles(M', ts_c), \\
M' &= fold(set, M, us), \\
us &= aggregate_branches(M, ts_b), \\
ts_b &= ts \setminus ts_c, \\
ts_c &= ((i,j),t) \ . \ ((i,j),t) \in ts, i = j, \\
ts &= aggregate_sequences(M, i)
\end{aligned}
$$

Fig. B.1. The $aggregate_smp$ function

This appendix describes more fully the semi-Markov process state-level aggregation algorithm presented by Bradley in [21]. We define the aggregation function, $aggregate_smp$, which is defined on an SMP, M, for a state i being aggregated. We represent M by the tuple (S, P, L) where S is the finite set of states, P is the underlying DTMC and L is the state holding-time distribution matrix whose entries are the Laplace transforms of the state sojourn-time densities. In an irreducible semi-Markov process, the aggrega-

tion procedure can be applied to any state, while in a transient SMP, it may be applied to any state except the absorbing or initial states. Reading from the bottom up in the function definition of Fig. B.1:

1. $ts = aggregate_sequences(M, i)$:

   Find all valid paths of length two that have i as the centre state. Form a set of single transitions using the sequential aggregation described in Section 5.2.1.

2. $ts_b = ts \setminus ts_c,\ ts_c = \{((i,j),t)\ .\ ((i,j),t) \in ts, i = j\}$:

   Separate the transition set ts into a set of cycles, ts_c, and a set of branching transitions to other states, ts_b.

3. $us = aggregate_branches(M, ts_b)$:

   Where transitions in M are duplicated by transitions in ts_b, these are aggregated and placed in us using the branching aggregation from Section 5.2.1. Where a new transition that is not present in M in described ts_b then that transition is written unchanged into us.

4. $M' = fold(set, M, us)$:

   Overwrite all the transitions in M that are present in us.

5. $vs = aggregate_cycles(M', ts_c)$:

   Perform cyclic aggregation on the transitions in ts_c, using the method from Section 5.2.1.

6. $aggregate_smp(M, i) = fold(set, M', vs)$:

   Finally, overwrite the transitions of M' with the replacement vs transitions and return the final SMP.

The result is that M' will have a disconnected state, i, which can be removed from the transition matrices.

B.1 Aggregation Functions

This section defines the key functions used by $aggregate_smp$: $aggregate_sequences$, $aggregate_branches$ and $aggregate_cycles$. As before, we define the set SMP as the tuple (S, P, L). We also use $PD = (Prob, LaplaceT)$ to be a tuple of a probability and Laplace transform associated with a particular transition. $T = (S \times S) \times PD$, associates states i and j states to a given transition pair. We define $i \xrightarrow{M} j$ to mean that there exists a transition from state i to state j in M.

$aggregate_sequences$ takes a state to be aggregated, i, and the SMP and returns a set of transitions which represent all the paths (from one state preceding to one state after i) which pass through i. The paths have been aggregated into single transitions using agg_seq. The number of transitions generated is equal to the product the number transitions leading into i and the number leaving the state.

$$
\begin{aligned}
aggregate_sequences \quad &: \quad SMP \times S \rightarrow \mathcal{P}(T) \\
aggregate_sequences \quad &\quad (M, i) = \{((i, j), agg_seq(t, t')) \;.\; (i, t) \in trans_to(i, M), \\
&\qquad (j, t') \in trans_from(i, M)\}
\end{aligned}
$$

$aggregate_branches$ processes an SMP, M, and a set of transitions, R, which must not contain any cycles. The transitions are to be integrated into the SMP, M. If there is a pre-existing transition in M for a given member of R, then the two are combined using agg_branch, otherwise the transition in R is just returned unchanged.

$$
\begin{aligned}
aggregate_branches \quad &: \quad SMP \times \mathcal{P}(T) \rightarrow \mathcal{P}(T) \\
aggregate_branches \quad &\quad (M, R) = \{((i, j), agg_branch(t, t')) \;.\; ((i, j), t)) \in R, \\
&\qquad t' = \text{ if } i \xrightarrow{M} j \text{ then } (M_P(i, j), M_L(i, j)) \text{ else } (0, 0)\}
\end{aligned}
$$

$aggregate_cycles$ processes an SMP, M, and a set of cyclic transitions, R. For each cycle defined in R from i to i, the set of out-transitions is selected from the SMP, M. For each member of that set, the aggregation function agg_cycle is applied. All the

modified transitions are unified into a single set and returned.

$$\begin{aligned} aggregate_cycles &: SMP \times \mathcal{P}(T) \to \mathcal{P}(T) \\ aggregate_cycles & \quad (M, R) = \bigcup_{i:((i,i),t') \in R} X \\ & \quad . \; X = \{((i,j), agg_cycle(t, t')) \; . \; (j,t) \in trans_from(i, M)\} \end{aligned}$$

agg_seq is used to represent two transitions in sequence as a single transition.

$$\begin{aligned} agg_seq &: PD \times PD \to PD \\ agg_seq & \quad ((p, d(z)), (p', d'(z))) = (pp', d(z)d'(z)) \end{aligned}$$

agg_branch is used to represent two branching transitions which terminate in the same state as a single transition.

$$\begin{aligned} agg_branch &: PD \times PD \to PD \\ agg_branch & \quad ((p, d(z)), (p', d'(z))) = (p + p', \frac{p}{p+p'} d(z) + \frac{p'}{p+p'} d'(z)) \end{aligned}$$

agg_cycle is used to represent a cycle to the same state and a leaving transition as a single transition. The first argument is the leaving transition and the second is the cyclic transition. If there is more than one out-transition then the transformation will need to be applied to each in turn.

$$\begin{aligned} agg_cycle &: PD \times PD \to PD \\ agg_cycle & \quad ((p, d(z)), (p_c, d_c(z))) = \left(\frac{p}{1 - p_c}, \frac{(1 - p_c) d(z)}{1 - p_c d_c(z)} \right) \end{aligned}$$

B.2 Utility functions

The definitions of $trans_from$, $trans_to$, set, $fold$ and $elem_rest$:

The function $trans_from$ takes a state, i, and returns the set of states, probabilities and distributions which succeed i.

$$\begin{aligned} trans_from &: S \times SMP \to \mathcal{P}(S \times PD) \\ trans_from & \quad (i, M) = \{(j, (M_P(i,j), M_L(i,j))) \; . \; j \in M_S, i \xrightarrow{M} j\} \end{aligned}$$

The function $trans_to$ takes a state, i, and returns the set of states, probabilities and distributions which connect to i.

$$\begin{aligned} trans_to &: S \times SMP \rightarrow \mathcal{P}(S \times PD) \\ trans_to & \quad (i, M) = \{(j, (M_P(j,i), M_L(j,i))) \ . \ j \in M_S, j \xrightarrow{M} i\} \end{aligned}$$

The set function is used to set a given transition in the underlying DTMC and distribution matrix of an SMP. If necessary the current transition is overwritten.

$$\begin{aligned} set &: T \times SMP \rightarrow SMP \\ set & \quad (((i,j),(p,d(z))), M) = (M_S, P', L') \\ & \qquad . \ P'(k,l) = \begin{cases} p & \text{if } (k,l) = (i,j) \\ M_P(k,l) & \text{otherwise} \end{cases} \\ & \qquad L'(k,l) = \begin{cases} d(z) & \text{if } (k,l) = (i,j) \\ M_L(k,l) & \text{otherwise} \end{cases} \end{aligned}$$

$fold$ is used to add whole sets of new or replacement transitions to an SMP, as done in Fig. B.1.

$$\begin{aligned} fold &: (A \times B \rightarrow B) \times B \times \mathcal{P}(A) \rightarrow B \\ fold & \quad (f, r, \Gamma) = \begin{cases} r & \text{if } \Gamma = \emptyset \\ f(x, r, fold(f, r, \Gamma')) & \\ \qquad . \ (x, \Gamma') = elem_rest(\Gamma) & \text{otherwise} \end{cases} \end{aligned}$$

The function, $elem_rest$, is used by $fold$ to select an arbitrary element from a set and return a tuple containing that element and the set minus that element.

$$\begin{aligned} elem_rest &: \mathcal{P}(A) \rightarrow (A, \mathcal{P}(A)) \\ elem_rest & \quad (T) = \begin{cases} \perp & \text{if } T = \emptyset \\ (x \ . \ x \in T, \{t \ . \ t \in T, t \neq x\}) & \text{otherwise} \end{cases} \end{aligned}$$

Bibliography

[1] J. Abate, G. Choudhury, and W. Whitt. An introduction to numerical transform inversion and its application to probability models. In W. Grassman, editor, *Computational Probability*, pages 257–323, Kluwer, Boston, 2000.

[2] J. Abate, G.L. Choudhury, and W. Whitt. On the Laguerre method for numerically inverting Laplace transforms. *INFORMS Journal on Computing*, 8(4):413–427, 1996.

[3] J. Abate and W. Whitt. The Fourier-series method for inverting transforms of probability distributions. *Queueing Systems*, 10(1):5–88, 1992.

[4] J. Abate and W. Whitt. Numerical inversion of Laplace transforms of probability distributions. *ORSA Journal on Computing*, 7(1):36–43, 1995.

[5] M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.

[6] A. Argent-Katwala, J.T. Bradley, and N.J. Dingle. Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP'04)*, pages 49–58, Redwood Shores CA, USA, January 14th–16th 2004.

[7] S.W. Au-Yeung. Finding probability distributions from moments. Master's thesis, Imperial College, London, United Kingdom, 2003.

[8] S.W. Au-Yeung, N.J. Dingle, and W.J. Knottenbelt. Efficient approximation of response time densities and quantiles in stochastic models. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP'04)*, pages 151–155, Redwood Shores CA, USA, January 14th–16th 2004.

[9] Australian Capital Territory Commissioner for the Environment. Indicator: Emergency services, February 2004. URL: `http://www.environmentcommissioner.act.gov.au/SoE2000/ACT/IndicatorResults/Emergencyservices.htm`.

[10] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous-time Markov chains. In *Lecture Notes in Computer Science 1102: Computer-Aided Verification*, pages 269–276. Springer-Verlag, 1996.

[11] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[12] S. Baase and A. van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. Addison-Wesley, Reading, MA, 3rd edition, 2000.

[13] C. Baier, B.R. Haverkort, H. Hermanns, and J-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Lecture Notes in Computer Science 1855: Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, pages 358–372, Chicago IL, USA, 2000. Springer-Verlag.

[14] C. Baier, J-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Lecture Notes in Computer Science 1664: Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, pages 142–162. Springer-Verlag, 1999.

[15] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets – An Introduction to the Theory*. Verlag Vieweg, Wiesbaden, Germany, 1995.

[16] M. Benzi and M. Tuma. A parallel solver for large-scale Markov chains. *Applied Numerical Mathematics*, 41:135–153, 2002.

[17] C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[18] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland, Amsterdam, 1989.

[19] K. Blathras, D. Szyld, and Y. Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *Journal of Parallel and Distributed Computing*, 58(3):446–465, September 1999.

[20] G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, August 1998.

[21] J.T. Bradley. A passage-time preserving equivalence for semi-Markov processes. In *Lecture Notes in Computer Science 2324: Proceedings of the 12th International Conference on Modelling, Techniques and Tools (TOOLS'02)*, pages 178–187, London, April 14th–17th 2002. Springer-Verlag.

[22] J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Derivation of passage-time densities in PEPA models using ipc: the Imperial PEPA Compiler. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'03)*, pages 344–351, Orlando FL, USA, October 12th–15th 2003.

[23] J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Extracting passage times from PEPA models with the HYDRA tool: A case study. In *Proceedings of the 19th UK Performance Engineering Workshop (UKPEW'03)*, pages 79–90, Warwick, July 9th–10th 2003.

[24] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Distributed computation of passage time quantiles and transient state distributions in large Semi-Markov models. In *Proceedings of the International Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS'03)*, Nice, April 26th 2003.

[25] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Performance queries on semi-Markov stochastic Petri nets with an extended Continuous Stochastic Logic. In *Proceedings of 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, pages 62–71, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[26] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Distributed computation of transient state distributions and passage time quantiles in large Semi-Markov models. *Future Generation Computer Systems*, 2004. (to appear).

[27] J.T. Bradley, N.J. Dingle, and W.J. Knottenbelt. Strategies for exact iterative aggregation of semi-Markov performance models. In *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'03)*, pages 755–762, Montreal, Canada, July 20th–24th 2003.

[28] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. In *Proceedings of the 4th International Conference on the Numerical Solution of Markov Chains (NSMC'03)*, pages 99–120, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[29] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. *Linear Algebra and its Applications*, 386:311–334, 2004.

[30] J.T. Bradley and H.J. Wilson. Convergence and correctness of an iterative scheme for calculating passage times in semi-Markov processes. *Performance Evaluation*, 2004. (to appear).

[31] P. Buchholz. Hierarchical Markovian models: Symmetries and aggregation. *Performance Evaluation*, 22:93–110, 1995.

[32] P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In *Proceedings of the 3rd International Conference on*

the Numerical Solution of Markov Chains (NSMC'99), pages 76–95, Zaragoza, Spain, September 1999.

[33] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16:527–531, 1973.

[34] W-L. Cao and W.J. Stewart. Iterative aggregation/disaggregation techniques for nearly uncoupled Markov chains. *Journal of the ACM*, 32(3):702–719, July 1985.

[35] U.V. Catalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.

[36] U.V. Catalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool. Technical Report BU-CE-9915, Version 3.0, Department of Computer Engineering, Bikent University, Ankara, 06800, Turkey, 1999.

[37] G. Ciardo, M. Forno, P.L.E. Grieco, and A.S. Miner. Comparing implicit representations of large CTMCs. In *Proceedings of the 4th International Conference on the Numerical Solution of Markov Chains (NSMC'03)*, pages 323–327, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[38] G. Ciardo and A.S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proceedings of the 8th International Conference on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, September 1999. IEEE Computer Society Press.

[39] G. Ciardo, J. Muppala, and K.S. Trivedi. SPNP: Stochastic petri net package. In *Proceedings of the 3rd International Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 142–151, Kyoto, 1989.

[40] G. Ciardo, J.K. Muppula, and K.S. Trivedi. On the solution of GSPN reward models. *Performance Evaluation*, 12(4):237–253, 1991.

[41] G. Ciardo and K.S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[42] Commission for Health Improvement. Final key targets and performance indicators for primary care trusts (PCTs), December 2003. URL: http://www.chi.nhs.uk/eng/ratings/2004/index.shtml.

[43] County of Oxford Board of Health. 2001 annual report, October 2002. URL: http://www.county.oxford.on.ca/healthservices/ocbh/pdf/2001_Annual_Report_PublicHealth_Oct29_2002.pdf.

[44] Cross-Industry Working Team. Customer view of internet service performance: Measurement methodology and metrics, October 1998. URLs: http://www.xiwt.org/documents/IPERF-paper.pdf, http://www.metron.co.uk/reference/technical/tech31.doc.

[45] I. Davies, W.J. Knottenbelt, and P.S. Kritzinger. Symbolic methods for the state space exploration of GSPN models. In *Lecture Notes in Computer Science 2324: Proceedings of the 12th International Conference on Modelling, Techniques and Tools (TOOLS'02)*, pages 188–199, London, April 14th–17th 2002. Springer Verlag.

[46] D.D. Deavours and W.H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33(1):67–84, June 1998.

[47] D.D. Deavours and W.H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.

[48] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Response time densities in Generalised Stochastic Petri Net models. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP'02)*, pages 46–54, Rome, July 24th–26th 2002.

[49] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. HYDRA: HYpergraph-based Distributed Response-time Analyser. In *Proceedings of the International Con-*

ference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), pages 215–219, Las Vegas NV, USA, June 23rd–26th 2003.

[50] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.

[51] N.J. Dingle and W.J. Knottenbelt. Distributed solution of large Markov models using asynchronous iterations and graph partitioning. In *Proceedings of the 18th UK Performance Engineering Workshop (UKPEW'02)*, pages 27–34, Glasgow, July 10th–11th 2002.

[52] DLT Solutions Inc. Capacity planning for e-commerce systems with Benchmark Factory™, February 2004. URL: `http://www.dlt.com/quest/resources-whitepapers.asp`.

[53] H. Dubner and J. Abate. Numerical inversion of Laplace transforms by relating them to the finite Fourier cosine transform. *Journal of the ACM*, 15(1):115–123, 1968.

[54] D.G. Duffy. On the numerical inversion of Laplace transforms: comparison of three new methods on characteristic problems from applications. *ACM Transactions on Mathematical Software*, 19(3):333–357, September 1993.

[55] J.B. Dugan, K. Trivedi, R. Geist, and V. Nicola. Extended stochastic Petri nets: Applications and analysis. In *Proceedings of the 10th International Symposium on Models of Computer System Performance (Performance '84)*, pages 507–519, Paris, December 19th–21th 1984.

[56] P.P.G. Dyke. *An Introduction to Laplace Transforms and Fourier Series*. Springer-Verlag, 2001.

[57] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[58] J. Freiheit and A. Heindl. Novel formulae for GSPN aggregation. In *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, pages 209–216, Fort Worth TX, USA, October 11th–16th 2002.

[59] M. Freimer, G. Kollia, G.S. Mudholkar, and C.T. Lin. A study of the generalized Tukey Lambda family. *Communications in Statistics: Theory and Methods*, 17(10):3547–3567, 1988.

[60] R.M. Fricks, A. Puliafito, M. Telek, and K. Trivedi. Applications of non-Markovian stochastic Petri nets. *Performance Evaluation Review*, 26(2):15–27, 1998.

[61] A. Frommer and D.B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.

[62] M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representations. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[63] R. German. *Performance Analysis of Communication Systems: Modelling with Non-Markovian Stochastic Petri Nets*. John Wiley & Sons, 2000.

[64] R. German, D. Logothetis, and K.S. Trivedi. Transient analysis of Markov regenerative stochastic Petri nets: A comparison of approaches. In *Proceedings of the 6th International Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 103–112, Durham, North Carolina, 1995.

[65] S.T. Gilmore, J. Hillston, and G. Clark. Specifying performance measures for PEPA. In *Lecture Notes in Computer Science 1691: Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems*, Bamberg, 1999. Springer-Verlag.

[66] W. Grassman. Means and variances of time averages in Markovian environments. *European Journal of Operational Research*, 31(1):132–139, 1987.

[67] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachussetts, 1994.

[68] P.G. Harrison. Laplace transform inversion and passage-time distributions in Markov processes. *Journal of Applied Probability*, 27:74–87, 1990.

[69] P.G. Harrison. The MM CPP/GE/c G-Queue: sojourn time distribution. *Queueing Systems*, 41:271–298, January 2002.

[70] P.G. Harrison and W.J. Knottenbelt. Passage time distributions in large Markov chains. In *Proceedings of ACM SIGMETRICS 2002*, pages 77–85, Marina Del Rey, California, June 2002.

[71] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures*. International Computer Science Series. Addison Wesley, 1993.

[72] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the ACM/IEEE Supercomputing Conference*. ACM/IEEE, December 1995.

[73] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix–vector multiplication. *International Journal of High Speed Computing*, 7(1):73–88, 1995.

[74] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Proceedings of the 3rd International Conference on the Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207, Zaragoza, Spain, September 1999.

[75] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[76] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[77] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[78] Ng Chee Hock. *Queueing Modelling Fundamentals*. John Wiley and Sons, 1996.

[79] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[80] G. Karypis and V. Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*. University of Minnesota, September 1998.

[81] G. Karypis and V. Kumar. Multilevel k-way hypergraph paritititioning. Technical Report #98-036, University of Minnesota, 1998.

[82] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[83] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report #96-036, University of Minnesota, 1998.

[84] G. Karypis, K. Schloegel, and V. Kumar. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0*. University of Minnesota, September 1998.

[85] J-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *Lecture Notes in Computer Science 2165: Proceedings of Process Algebra and Probabilistic Methods (PAPM'01)*, pages 23–38, Aachen, September 2001. Springer-Verlag.

[86] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Van Nostrand, 1960.

[87] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.

[88] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, February 2000.

[89] W.J. Knottenbelt and P.G. Harrison. Distributed disk-based solution techniques for large Markov models. In *Proceedings of the 3rd International Conference on the Numerical Solution of Markov Chains (NSMC'99)*, pages 58–75, Zaragoza, Spain, September 1999.

[90] W.J. Knottenbelt, P.G. Harrison, M.A. Mestern, and P.S. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation*, 39(1–4):127–148, February 2000.

[91] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[92] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing, 1994.

[93] M. Kwiatkowska and R. Mehmood. Out-of-core solutions of large linear systems of equations arising from stochastic modelling. In *Proceedings of Process Algebra and Performance Modelling (PAPM'02)*, pages 135–151, Copenhagen, July 25th–26th 2002.

[94] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Lecture Notes in Computer Science 2324: Proceedings of the 12th International Conference on Modelling, Techniques and Tools (TOOLS'02)*, pages 200–204, London, April 14th–17th 2002. Springer Verlag.

[95] A. Lakhany and H. Mausser. Estimating the parameters of the Generalized Lambda Distribution. *Algo Research Quarterly*, 3(3):47–58, December 2000.

[96] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley and Sons, 1998.

[97] London Ambulance Service. Category A response times, February 2004. URL: http://www.londonambulance.nhs.uk/news/performance/performance.html.

[98] G.G. Infante López, H. Hermanns, and J-P. Katoen. Beyond memoryless distributions: Model checking semi-Markov chains. In *Lecture Notes in Computer Science 2165: Proceedings of Process Algebra and Probabilistic Methods (PAPM'01)*, pages 57–70, Aachen, September 2001. Springer-Verlag.

[99] B. Melamed and M. Yadin. Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes. *Operations Research*, 32(4):926–944, July–August 1984.

[100] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[101] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[102] A.S. Miner. Computing response time distributions using stochastic Petri nets and matrix diagrams. In *Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, pages 10–19, Urbana-Champaign, IL, September 2nd–5th 2003.

[103] I. Mitrani. *Probabilistic Modelling*. Cambridge University Press, August 1998.

[104] Municipal Corporation of the County of Renfrew. Health committee minutes, May 2003. URL: http://www.countyofrenfrew.on.ca/2003/Health/Minutes/HMinutes%20May%2014-03.pdf.

[105] J.K. Muppala and K.S. Trivedi. Numerical transient analysis of finite Markovian queueing systems. In U.N. Bhat and I.V. Basawa, editors, *Queueing and Related Models*, pages 262–284. Oxford University Press, 1992.

[106] M.F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*. Johns Hopkins University Press, Baltimore, MD, 1981.

[107] F. Oberhettinger and L. Badii. *Tables of Laplace Transforms*. Spring-Verlag, 1973.

[108] A.T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, May 1993.

[109] A. Puliafito, M. Scarpa, and K.S. Trivedi. Petri nets with k simultaneously enabled generally distributed timed transitions. *Performance Evaluation*, 32(1):1–34, 1998.

[110] R. Pyke. Markov renewal processes with finitely many states. *Annals of Mathematical Statistics*, 32(4):1243–1259, December 1961.

[111] S. Rácz. *Numerical Analysis of Communication Systems Through Markov Reward Models*. PhD thesis, Budapest University of Technology and Economics, 2002.

[112] A. Reibman and K.S. Trivedi. Numerical transient analysis of Markov models. *Computers and Operations Research*, 15(1):19–36, 1988.

[113] San Francisco EMS Section Department of Public Health. San Francisco EMS system activity summary, December 1999. URL: `http://www.sanfranciscoems.org/publication/SystemActivities/emssystemactivities1999.pdf`.

[114] M. Sczittnick. Technicken zur funktionalen und quantitativen Analyse von Markoffschen Rechensystemmodellen. Diplomarbeit, Universität Dortmund, October 1987.

[115] R.M. Simon, M.T. Stroot, and G.H. Weiss. Numerical inversion of Laplace transforms with application to percentage labeled mitoses experiments. *Computers and Biomedical Research*, 5(6):596–607, 1972.

[116] W.J. Stewart. MARCA: Markov chain analyser. a software package for Markov modelling. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*, pages 37–62. Marcel Dekker Inc., New York, 1991.

[117] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[118] A. Talbot. The accurate numerical inversion of Laplace transforms. *Journal of the Institute of Mathematical Applications*, 23:97–120, 1979.

[119] Township of Rideau Lakes. Leeds Grenville emergency medical services frequently asked questions, February 2004. URL: `http://www.twprideaulakes.on.ca/ambulance-faq.html`.

[120] Transaction Processing Performance Council. TPC benchmark C: Standard specification revision 5.2, December 2003. URL: `http://www.tpc.org/tpcc/default.asp`.

[121] A. Trifunovic and W.J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC'04)*, University College Cork, Ireland, July 5th–7th 2004.

[122] A. Trifunovic and W.J. Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS'04)*, Antalya, Turkey, October 27th–29th 2004.

[123] A. Trifunovic and W.J. Knottenbelt. Towards a parallel algorithm for multilevel k-way hypergraph partitioning. In *Proceedings of the 5th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'04)*, Santa Fe NM, April 26th–30th 2004.

[124] W.T. Weeks. Numerical inversion of Laplace transforms using Laguerre functions. *Journal of the ACM*, 13(3):419–429, 1966.

[125] C.M. Woodside and Y. Li. Performance Petri net analysis of communication protocol software by delay-equivalent aggregation. In *Proceedings of the 4th International Workshop on Petri nets and Performance Models (PNPM'91)*,

pages 64–73, Melbourne, Australia, 2–5 December 1991. IEEE Computer Society Press.

[126] H.L.S. Younes and R.G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Lecture Notes in Computer Science 2404: Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, pages 223–235, Copenhagen, July 2002. Springer-Verlag.