Performance Trees: Implementation and Distributed Evaluation

Author: Darren K. Brien
dkb03@doc.ic.ac.uk

Project Supervisor: William J. Knottenbelt
wjk@doc.ic.ac.uk

June 18, 2008
Abstract

Performance Trees are a graphical performance query specification formalism that enables the expression of complex queries containing both performance requirements and performance measures. They support a wide range of concepts that are likely to be familiar to system designers and performance engineers alike. In this report we present an architecture for a parallel and distributed evaluation pipeline for Performance Tree queries that comprises a client-side model and performance query specification tool, and a server-side distributed evaluation engine, supported by a dedicated computing cluster. The evaluation engine combines the analytic capabilities of a number of distributed tools for steady-state and passage time analysis, and also incorporates a caching mechanism to avoid redundant calculations. We demonstrate the expressive power of Performance Trees through a case study evaluation of a hospital’s Accident & Emergency unit. This shows how the analysis pipeline allows remote users to design their models and performance queries in a sophisticated yet easy-to-use framework, evaluate them by harnessing the computing power of a Grid cluster back-end, and subsequently assess results through an intuitive user interface.
Acknowledgements

I would like to acknowledge the following people for their help and guidance during the course of this project:

- Dr William Knottenbelt, for being an enthusiastic and insightful project supervisor, available at a moments notice to discuss my work and for allowing a lowly Masters student gain his first publication.

- Tamas Suto, for allowing me to work on this stimulating project related to his PhD, for his help introducing me to PIPE tool and providing an excellent basis upon which to begin this project.

- Nicolas Dingle and Harini Kulatunga, for introducing me to the Analysis Tools, specifically SMARTA, and for their guidance in integrating these into my evaluation architecture.

- The Computing Support Group (CSG), especially Matt Johnson for his timely responses to my issues relating to the Camelot cluster.

- My many friends, who over the last year got me away from my desk for sufficient periods of time to provide a different perspective on my work.
Dedication

To my loving parents who have been a source of unconditional support and encouragement throughout my studies.

Thank you for giving me the opportunity to study at a world class institution, I got there in the end, with your help I knew we always would.
# Contents

1 Introduction ............................................. 1  
1.1 Motivation and Objectives .......................... 1  
1.2 Contributions ........................................ 3  
1.3 Publications .......................................... 5  

2 Background ............................................. 7  
2.1 Stochastic Modelling ................................. 7  
  2.1.1 Introduction ....................................... 7  
  2.1.2 Markov & Semi Markov processes .............. 8  
  2.1.3 Generalized Stochastic Petri Nets - GSPNs .... 8  
  2.1.4 Stochastic process algebras .................... 10  
  2.1.5 Queueing Networks .............................. 10  
2.2 Extracting Performance properties from Stochastic Models .......................... 11  
  2.2.1 Laplace Transform ............................... 11  
  2.2.2 Steady-State Calculations for GSPNs ........... 12  
  2.2.3 Passage Time Density Calculations for GSPNs ... 12  
  2.2.4 Moments Calculation for GSPNs ................. 14  
2.3 Performance Trees ................................... 14  
  2.3.1 Introduction ..................................... 14  
  2.3.2 Motivation ...................................... 15  
  2.3.3 Operations Available ........................... 15  
2.4 Performance Measures and Performance Requirements .......................... 15  
2.5 Stochastic Logics ................................... 17  
  2.5.1 CSL ............................................. 17  
  2.5.2 aCSL ........................................... 18  
  2.5.3 eCSL ........................................... 18  
  2.5.4 CSL compared to Performance Trees .......... 18  
2.6 Existing Evaluation Tools ........................... 19
3 Architecture
3.1 Pre-existing Performance Analysis Architecture
3.2 New Architecture
3.3 Client-Side Application
  3.3.1 GSPN Editor Module
  3.3.2 Performance Query Editor Module
  3.3.3 Performance Tree Evaluation Module
3.4 Server-Side Application
  3.4.1 Analysis Server
  3.4.2 Analysis Tools
  3.4.3 Cache verification
  3.4.4 Camelot Cluster
4 Implementation
4.1 Evaluation of performance trees with dependent nodes
  4.1.1 The Problem
  4.1.2 Multi-Threaded Evaluation and Java™
  4.1.3 Callable interface
  4.1.4 Future interface
  4.1.5 ThreadPool class
  4.1.6 ExecutorService interface
  4.1.7 Supporting Classes
  4.1.8 Bringing it all together
4.2 Translation of Performance Tree to Tool Specific Languages
4.3 Client-Server Network Interaction
4.4 Logging System
4.5 Result Display Module
4.6 Camelot Cluster Configuration
  4.6.1 DRMAA Singleton
  4.6.2 Processor allocation
4.7 Performance Tree Node Evaluation
  4.7.1 DNAmaca: Steady state Analyzer
  4.7.2 SMARTA: Passage Time Analyzer
  4.7.3 MOMA: Moment Analysis
  4.7.4 Moments of Steady State probability distributions
  4.7.5 PERC: PERCentile Analysis
  4.7.6 PROBI: PROBability in Interval Analysis
  4.7.7 CONE: CONvolution of passagE time densities
  4.7.8 Boolean and Arithmetic Analysis
## CONTENTS

4.7.9 State Function specification verification .................................. 45

5 Case Study ............................................................................. 47
5.1 Accident and Emergency Model ............................................. 47
5.2 Example 1 ................................................................. 48
5.3 Example 2 ................................................................. 50

6 Evaluation ............................................................................ 53
6.1 Comparison with Existing Evaluation Tools ......................... 53
6.2 Assessment of Functionality ............................................... 54
6.3 Benchmarking ..................................................................... 54
   6.3.1 System performance with multiple clients ..................... 54
   6.3.2 System performance with varying processor allocation .... 55
   6.3.3 Comparison with old architecture ................................. 56
6.4 Summary ............................................................................. 57

7 Conclusion ............................................................................ 59
7.1 Summary of Achievements .................................................. 59
7.2 Discussion of Applications .................................................. 59
7.3 Further work ....................................................................... 60
   7.3.1 Support for all Performance Tree Operation Nodes ......... 60
   7.3.2 Further optimisations for Performance Tree evaluation ... 60
   7.3.3 Natural Language Query Specification ......................... 61
   7.3.4 Tagged Customer Support ........................................... 61
   7.3.5 SPA model extension .................................................. 62

A Extra Case Study Screen shots .................................................. 63

Bibliography .............................................................................. 64
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example GSPN model of a Hospital A&amp;E Department</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Old Evaluation Architecture</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Performance Tree Evaluation Architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Performance Query Editor module showing a performance requirement verification query</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluation window displaying PT with $PTD$ and $Percentile$ nodes, yellow indicators show evaluation is in progress</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Example 1 Performance Tree demonstrating node dependencies</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Example 2 Performance Tree demonstrating node dependencies</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Class Diagram of the Analysis Server and related components</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Sequence Diagram describing Performance Tree Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>GSPN model of a Hospital A&amp;E Department</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance Tree query with $PTD$ and $Percentile$ nodes</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Case Study Example 1: screen-shots of Performance Tree Evaluation Module</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Steady state and firing rate measure specification using $Sequential$ operator for multiple queries</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>Case Study Example 2: screen-shots of Performance Tree Evaluation Module</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph showing time taken to evaluate a query with multiple clients</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Graph showing time taken to evaluate a query with varying processor allocation</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Example Performance Tree for coefficient of variation</td>
<td>61</td>
</tr>
<tr>
<td>A.1</td>
<td>Case Study Example 2 screen-shots of Performance Tree Evaluation Module</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation and Objectives

Systems in many varying domains - including defence, healthcare, logistics, telecoms and transport - are required to conform to end user reliability and performance requirements [35,36]. The importance of such requirements cannot be understated because, more often than not, if they remain unsatisfied, a system will be left seriously debilitated in fulfilling its prescribed purpose. For example, a telecommunications system that is unable to guarantee successful and timely delivery of SMS messages in 95% of cases significantly degrades one's perception of network service. Through observation and analysis of the network, the parameters that govern the predominant behavior can be identified, allowing one to reason about why service criteria are not being met.

It may be possible to obtain measurements of performance through observation, by monitoring a system over a reasonable period of time and obtaining performance statistics. There are several shortcomings to this approach: it can be time consuming, it may be difficult to implement and a “reasonable period of time” is arbitrary. Furthermore, this approach does not apply well to analyzing systems in the design phase. Therefore engineers are required to construct and analyze performance models to allow them to analyse their systems.

Many formalisms exist to model a system’s performance characteristics these include queuing networks, stochastic Petri nets and stochastic process algebras (see Section 2.1). These high-level formalisms can be mapped onto underlying Continuous Time Markov chains (CTMC) which describe the set of states a system may be in at a particular time and the transitions between states that can be made. With the CTMC extracted, if a numerically feasible number of states exist the performance characteristics of the system can be extracted. These modelling formalisms allow performance analysis to be carried out in a well defined mathematical framework using performance queries to articulate the particular performance characteristics required by the system engineer.

Performance queries provide a mechanism with which a model’s performance characteristics can be expressed and verified. Generally the kind of performance characteristics that we may wish to obtain from a model fall into two categories.
1.1. Motivation and Objectives

**Performance measures**, quantitative measures of the properties of the system model, for example

- In Retail - **What is the steady state distribution of people waiting to be served at a supermarket check-out?**

These measures are typically expressed in the tool-specific languages of quantitative analyzers [36], these tend to be designed to support ease of interpretation by a software tool programatically. This means that the languages tend to be less accessible to the system engineer expressing the measure.

**Performance requirements**, state a stochastic property of a system which must be satisfied, for example

- In Finance - In a trading system, 80% of trades should be settled in less than 3 days.

These measures are typically expressed in stochastic logical formulae and are evaluated by model checkers [36]. Logical formulae provide a concise and elegant way to express performance characteristics, however, there are limitations in their expressiveness and despite their prevalence among academics they can be enigmatic for systems engineers. There is support for this in [24] where in 154 examples of probabilistic formulae there were two cases where syntactically incorrect formulae were used, “which provides evidence that the correct specification for probabilistic properties is challenging, even for experts” (Gruniske [24] Section 3.2).

Clearly there is a disparity between the specification of both types of performance characteristics. System engineers may often wish to specify both performance measures and performance requirements on the same model, this would currently require the usage of two types of formalism. Therefore, a single formalism with the ability to specify both types of performance characteristics in a simple and intelligible way would overcome the shortcomings exhibited by current techniques.

In [35, 36] **Performance Trees** (PTs) were proposed as a new formalism in which both of the above types of properties can be specified in a familiar tree like structure, they support ease of use through a graphical specification mechanism. A suite of operators are provided, forming the nodes of the Performance Tree; these provide familiar functionality to system engineers and do not suffer some of the limitations of stochastic logical formulae in regard to their expressiveness (see Section 2.5 for an in depth evaluation of other formalisms in comparison to Performance Trees).

Several suites of performance analysis tools are available that to varying degrees support:

- Specification of system models.
- Composition of performance queries.
- Evaluation of a model based on a specified performance query.

Depending on the modelling and query formalisms used, expressive power and ease of use vary greatly between the available tools. Also as one may expect there is a separation between tools that support the specification of performance measures and performance requirements (see Section 2.6).
Performance Trees represent an attractive alternative to other formalisms and there is no existing performance analysis tool which unequivocally meets the requirements of systems engineers. A Performance Tree evaluation environment would be a valuable addition to the state of the art and would establish the first tool support for this new formalism. Such an environment would consist of:

- A client-side model and Performance Tree specification tool
- A server-side evaluation engine
- A dedicated computing cluster with the requisite tools to perform analysis on the model

At the outset of this project the open-source PIPE: Petri net editor [1] and related cluster based tools possessed some of the requisite functionality a distributed evaluation environment would require, namely:

- A system model specification tool, a GUI allowing for the graphical specification of a Petri net model.
- A Performance Tree specification tool, a GUI allowing for the graphical specification of a Performance Tree.
- SMARTA: a cluster based tool allowing for the calculation of passage time densities on Petri net models, results from this tool are returned to the remote client for representation inside PIPE.
- DNAmaca: a cluster based tool allowing for steady state calculations on Petri net models and results from this tool are returned to the remote client for representation inside PIPE.
- MOMA: a cluster based tool allowing for Moment calculations to be performed Petri net models.

By utilising existing functionality in PIPE, with the addition of new client and server side modules to support the evaluation of the full suite of Performance Tree operators, an evaluation environment for Performance Trees has been realised. This project contributes towards the GRAIL (Grid-based Performance Analysis using Stochastic Logics) GPSRC research grant.

1.2 Contributions

This project seeks to provide the necessary tool based support for the specification and evaluation of Performance Trees as laid out in Figure 3.2. There are four significant contributions made by this project.

1. Working with PIPE a tool with a code base of over 80,000 lines of code and adding and adapting modules in this tool where required.

2. Working with a variety of back-end tools and implementing new ones to support evaluation of the full suite of performance tree operators.
1.2. Contributions

3. Extending and refining the Performance Tree formalism, including the introduction of a new operator.

4. Providing an architecture supporting the evaluation of performance trees and visualisation of results.

The implementation specific contributions of this project are laid out below.

1. The ability to submit a Performance Tree from the PIPE Performance Tree Query Editor to the remote computational back-end for analysis to be performed; upon completion of analysis results are displayed. The work performed on the client-side includes:

- **Client-Server Architecture Enhancements**: Revision to the existing implementation supporting transmission of more varied data.

- **Evaluation Module GUI**: A window interface was developed to report on the progress of evaluation of a Performance Tree query and represent results in varying forms as they become available.

- **Enhancements and adjustments to existing tools**: The existing PIPE tool was adapted during this project, work included code maintenance, code refactoring and enhancing the object oriented structure of the code to facilitate better task encapsulation.

2. Co-ordinating support for the evaluation of all major Performance Tree nodes in the Computational back-end (see Table 2.1 for the full suite of Performance Tree nodes). The set of cluster based tools which were integrated into the performance tree evaluation architecture include:

- **PERC**: a tool allowing for the calculation of the percentile of a passage time distribution. This tool provides support for the **Percentile** Performance Tree node, a new Performance Tree node introduced to the formalism during the course of this project.

- **PROBI**: a tool allowing for the calculation of the probability a passage takes place in a certain amount of time. This tool provides support for the **Probability in Interval** Performance Tree node.

- **CONE**: a tool performing the convolution of two passage time densities. This tool provides support for the **Convolution** Performance Tree node.

- **MOMA**: a tool for the calculation of raw moments on a passage time density. *Adjustments* to this tool were made to provide support for the **Moment** Performance Tree node.

- **Moments of SS:P**: a tool to calculate raw moments of steady state probability distributions was implemented, along with **MOMA** this tool provides support for the **Moment** Performance Tree node.

- **SMARTA**: a tool for the calculation of passage time densities to provide support for the **Passage Time Density (PTD)** Performance Tree node. *Adjustments* to this tool to were made so that it worked in conjunction with the PROBI, PERC and CONE tools. This tool also provides support for the **Distribution** Performance Tree node, by providing the cumulative density function distribution points.
1. Introduction

Trivial Analysis Tools: a suite of tools that implement a recursive suite of procedures allowing for the evaluation of the Arithmetic comparison, Arithmetic operation, Boolean negation, Boolean disjunction, Boolean conjunction and In Interval Performance Tree nodes.

3. The design and implementation of an Analysis Server running on the server-side which co-ordinates evaluation between the client-side and Analysis tools. This module provides the following functionality:

Division of the Performance Tree into subtrees representing the individual calculations to be performed.

Dependency Analysis is performed on subtrees to establish an order to evaluate nodes ensuring correctness and efficiency.

Translation of each individual subtree into the tool specific language of the cluster based tools.

Dissemination of subtrees to the relevant Analysis Tools.

Retrieval of results from the Analysis Tools via the file system when evaluation has completed.

Result Transmission from server to client as results become available.

Logging of operations occurring on the Analysis Server to formulate a runtime reporting system allowing for fast diagnosis of problems.

1.3 Publications

The following publications arose from work conducted during the course of this project. The Chapters in this report where material from publications appear are indicated below.

• 7th International Workshop on Parallel and Distributed Methods in Verification (PDMC’08) [17] presents details of the architecture and implementation of an evaluation environment for Performance Trees that comprises a client-side model and performance query specification tool, and a server-side distributed evaluation engine, supported by a dedicated computing cluster. Demonstration of the environment is provided with a case study of a hospital Accident and Emergency unit. Material from this paper can be found in Chapter 2 on page 7, Chapter 4 on page 31 and Chapter 5 on page 47.

• 5th International Conference on the Quantitative Evaluation of Systems (QEST’08) [16] is a tool paper laying out the architecture of the evaluation environment including screen shots of the PIPE Petri net editor front-end. This paper includes details of updated and new tools provided in the architecture discussed in [17]. Material from this paper can be found in Chapter 3 on page 21.

• IEEE Transactions on Software Engineering (TSE) (submitted for publication) [18] details progressions since the publication of [16, 17] and demonstrates the expressive power of Performance Trees and of their application in the context of stochastic process algebras. Detailed discussion of the architecture and implementation of the evaluation environment are provided, including the
1.3. Publications

PIPE Petri net editor front-end and the Grid based back-end that integrates a wide range of performance analysis tools. A case study using an updated version of the hospital Accident and Emergency unit model is provided. Material from this paper can be found in Chapter 2 on page 7 and Chapter 4 on page 31.
Chapter 2

Background

This Chapter provides an overview of background theory in the area of Performance Analysis specific to the requirements of this project. The following topics are discussed:

- An introduction to Stochastic modelling and surrounding theory.
- Techniques used for extracting performance properties from Stochastic Models.
- An introduction to Performance Trees, their expressiveness and quantitative semantics.
- A comparison of Performance Trees with other commonly encountered performance query formalisms.
- An overview of some existing Performance Analysis tools.

2.1 Stochastic Modelling

2.1.1 Introduction

Varying formalisms are used to construct performance models. Naively one could argue that a single formalism alone could identify all possible states a system could enter and all the transitions that could occur between states. To specify every state and transition of even a moderately complex model would take so long as to make the process infeasible. For this reason, higher level formalisms are used to specify systems, leaving their underlying stochastic processes to be extracted and mapped onto Markov or semi-Markov chains so analysis can be performed. Stochastic Petri nets are one such graphical formalism used for the specification of performance models, the PIPE Petri net editor has a graphical interface for their design and manipulation, they are the modelling formalism used to specify system models in our evaluation architecture. For contrast a brief introduction to two other modelling formalisms, *Queuing Networks* and *Stochastic Process Algebras*, is provided.
2.1. Stochastic Modelling

2.1.2 Markov & Semi Markov processes

The lowest level representation of a performance modelling system is the state transition level. This is simply a collection of all the possible states that the system can enter and a collection of all the transitions which can be made between those states.

A stochastic process is an indeterminate process in which the path taken into the future is described by probability distributions. If the random variable \( X \) takes on different values at time \( t \) then \( X(t) \) is a stochastic process. The state space of \( X(t) \) is the set of different values which can be taken on by the process. If the state space of \( X(t) \) formulates a countable set then it is said to be discrete. If the state space of \( X(t) \) is uncountable then it is said to be continuous. Furthermore, if the different values of \( t \) that can be observed are countable then then we have a discrete time stochastic process, otherwise \( t \) is free to take any values and so we have a continuous time stochastic process.

A Markov process is a stochastic process in which the probability distribution of the next state depends only on the current state irrelevant of the path taken to enter it. The Markov property is stated as

\[
P[x(t_n) \leq x_n | x(t) \forall t \leq t_{n-1}] = P[x(t_n) \leq x_n | x(t_{n-1})]
\]  

(2.1)

A Markov process 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. [13].

A continuous-time Markov process is a stochastic process \( (X(t) : t \geq 0) \) which satisfies the Markov property. In a given state at time \( t \), the state of the process at time \( s \) is independent of the path of the process before \( t \).

Semi Markov processes are processes in which state transitions are determined by the probability distribution of a Markov process. However, time spent in a particular state (also known as Sojourn time) is described by a random variable dependent on the current state and the next state.

2.1.3 Generalized Stochastic Petri Nets - GSPNs

Stochastic Petri nets are the primary modelling formalism upon which the tools implemented during this project operate. This Section provides a brief summary of Stochastic Petri nets and surrounding theory in relation to Performance Trees. For a rigorous introduction to Stochastic Petri nets and surrounding theory the reader is directed to [13].

Petri nets were originally devised as a graphical formalism for describing concurrency and synchronisation in distributed systems. In their simplest form (not containing any timing information) they are also known as Place-Transition nets [13].

**Definition 2.1.1** A Place-Transition net is a 5-tuple

\[PN = (P,T,I^-,I^+,M_0)\]  

where:

- \( P = \{p_1, \ldots, p_n\} \) is a finite and non-empty set of places.
- \( T = \{t_1, \ldots, t_m\} \) is a finite and non-empty set of transitions.
- \( P \cap T = \emptyset \)
2. Background

2.1. Stochastic Modelling

- \( I^-, I^+ : P \times T \rightarrow \mathbb{N}_0 \) are the backward and forward incidence functions, respectively.
- \( M_0 : P \rightarrow \mathbb{N}_0 \) is the initial marking.

A marking (or state) is a vector of integers representing the number of tokens on each place of the model. A transition can fire if the input places of the transition contain at least the number of tokens specified by the backward incidence matrix. In so firing, a number of tokens are removed from the transition’s input places and a number of tokens added to the transition’s output places according to the backward and forward incidence matrices respectively.

Generalised Stochastic Petri Nets (GSPNs) extend Place-Transition nets by incorporating timing information \([7, 13]\).

**Definition 2.1.2** A GSPN is a 4-tuple \( GSPN = (PN, T_1, T_2, W) \) where

- \( PN = (P, T, I^-, I^+ M_0) \) is the underlying Place-Transition net.
- \( T_1 \subseteq T \) is the set of timed transitions.
- \( T_2 \subset T \) is the set of immediate transitions, where \( T_1 \cap T_2 = \emptyset \) and \( T = T_1 \cup T_2 \).
- \( W = (w_1, \ldots, w_{|T|}) \) is an array whose entry \( w_i \in \mathbb{R}^+ \) is a (possibly marking dependent)
  - rate of a negative exponential distribution specifying the firing delay, when transition \( t_i \) is a timed transition, or
  - firing weight, when transition \( t_i \) is an immediate transition.

Timed transitions have an exponentially distributed firing rate \( \lambda_i \). Immediate transitions fire in zero time. Markings that only enable timed transitions are known as tangible, while markings that enable both timed and immediate transitions are called vanishing. We denote the set of tangible markings \( T \) and the set of vanishing markings \( V \).

Figure 2.1: Example GSPN model of a Hospital A&E Department
2.1 Stochastic Modelling

2.1.2 The sojourn time in a tangible marking $M_i$ is exponentially distributed with parameter $\mu_i = \sum_{k \in \text{en}(M_i)} \lambda_k$ where $\text{en}(M_i)$ is the set of transitions enabled by marking $M_i$. The sojourn time in vanishing markings is zero.

An important property of stochastic Petri nets (SPNs) is that the reachability graph of the underlying Markov Chain is isomorphic. This means the number of states and the structure by which these states are connected is the same in both representations. Therefore properties are consistent between SPNs and Markov chains, this allows us to analyse models by performing steady-state, transient and passage time analysis on the CTMC. The same is true of GSPNs, however, vanishing states $^1$ must be eliminated which removes some of the fundamental properties of the original GSPN.

2.1.4 Stochastic process algebras

Process algebras are formalisms that provide the means to specify and reason about concurrent system behaviour. Actions are the fundamental building blocks that are used to describe entities that run concurrently and cooperate through communication. System models are constructed from smaller subsystems, which are composed to obtain a model that reflects the structure of the system under consideration. In pure process algebras, time is abstracted away within a process so that all actions are assumed to be instantaneous. To model real time behaviour, time may be represented by allowing an agent to witness periods of delay in addition to witnessing actions. Process algebras are often used to model systems in which there is uncertainty about the behaviour of a component. Probabilistic extensions of process algebras allow this uncertainty to be quantified because non-deterministic choice is replaced by probabilistic choice, where a probability is associated with each possible outcome of a choice.

One of the most widespread probabilistic process algebras is PEPA, the Performance Evaluation Process Algebra [29]. In PEPA, systems are described as interactions of components that can perform a set of activities. From a PEPA model it is possible to generate a reachability graph isomorphic to a CTMC; this allows us to perform steady-state, transient and passage time analysis in exactly the same manner as with a CTMC derived from a Petri net.

2.1.5 Queueing Networks

A further high-level modelling formalism is queueing networks [13, 25, 34]. Queueing networks are built from three basic components:

- **Servers** process customers and may have one or more queues connected to them. The time taken for a server to process a customer is a random variable.

- **Queues** store customers in the order that they arrive at a server; they can be fixed or infinitely large.

- **Customers** move between queues and are processed by servers; customers can be of different classes defining how they interact with other components in the network.

$^1$states with a sojourn time of zero
Queueing networks can be classed as either *open* or *closed*. When the population of customers in the network is fixed\(^2\) the network is closed. Otherwise, it is said to be open.

For closed queueing networks with Markovian service times, it is again possible to generate a reachability graph which is isomorphic to a CTMC. This can then be analysed for steady-state, transient and passage time quantities in exactly the same way as a CTMC derived from a Petri net or SPA model.

### 2.2 Extracting Performance properties from Stochastic Models

#### 2.2.1 Laplace Transform

The Laplace transform is an integral transform that is widely used in the solution of problems which are hard to solve in (real-valued) \(t\)-space. It transforms such problems into (complex-valued) \(s\)-space where they can be solved more easily; this solution is then inverted to bring it back into \(t\)-space.

The Laplace transform is a *linear* operator on a function \(f(t)\) where \(t\) is in real valued space, that transforms it to a function \(F(s)\) where \(s\) is in complex space.

\[
F(s) = \int_{0}^{\infty} f(t) e^{-st} dt
\]  

(2.2)

this is typically used to find the solution to differential equations which, for various reasons, may be difficult to solve in real, \(t\) space. The Laplace transform is defined for all functions \(f(t)\) of exponential order. As for our purposes we use the Laplace transform to find solutions to probability density functions (which by definition have a total integral of 1) the Laplace transform will always be defined.

A thorough introduction to the Laplace transform is outside of the scope of this report, however, as an example of its usage with respect to Performance Trees the *convolution operation* and *cumulative distribution function* are discussed below.

A useful property of the Laplace transform is that the convolution operation in the time domain is represented by the product in the Laplace domain.

\[
f(t) = \int_{0}^{t} u(\tau)v(t-\tau)d\tau \quad \text{then} \quad f^*(s) = u^*(s)v^*(s)
\]

(2.3)

We make use of this property when performing passage time density calculations on GSPN models (see Section 2.2.3).

A further appealing property of the Laplace transform is that a cumulative distribution function can be easily extracted by dividing the Laplace transform of the corresponding probability density function by \(s\). Higher moments are also simply obtained. If \(f(t)\) is a probability density function of a continuous random variable \(X\), then the \(n\)th moment of \(X\) is given by:

\[
\mathbb{E}(X^n) = (-1)^n f^{*(n)}(0)
\]

(2.4)

\(^2\)i.e. customers in the network cannot leave the network and new customers cannot arrive
2.2. Extracting Performance properties from Stochastic Models

**Laplace Inversion**

The inversion of a Laplace transform is a linear on a function $F(s)$ where $s$ is in complex space, which transforms it to a function $f(t)$ in real space. As detailed above one of the uses of the inverse Laplace transform in the scope of Performance trees is bring two convoluted PDFs back into real valued $t$ space. The equation for the inverse of the Laplace transform is (also known as the Bromwich integral)

$$f(t) = \frac{1}{2\pi i} \int_{\gamma}^{+\infty} e^{st} F(s) ds$$

(2.5)

where $\gamma$ is a real number greater than all the singularities of $f(s)$. This technique can be implemented using existing numerical methods, including the Laguerre and Euler methods, (details of these are provided in [22], where the methods used in the SMARTA tool of Section 4.7.2 are described).

**2.2.2 Steady-State Calculations for GSPNs**

The stochastic process described by a GSPN's reachability graph is a continuous-time Markov chain (CTMC) if $V = \emptyset$ and semi-Markovian otherwise. It is possible, however, to reduce the reachability graph of a GSPN where $V \neq \emptyset$ to a CTMC using vanishing-state elimination techniques [19, 31].

A homogeneous $N$-state CTMC has state at time $t$ denoted $\chi(t)$. Its evolution is described by an $N \times N$ generator matrix $Q$, where $q_{ij}$ is the infinitesimal rate of moving from state $i$ to state $j$ ($i \neq j$), and $q_{ii} = -\sum_{i \neq j} q_{ij}$.

Where it exists, the steady-state distribution of CTMC, $\{\pi_j\}$, is given by [13]:

$$\pi_j = \lim_{t \to \infty} \mathbb{P}(\chi(t) = j | \chi(0) = i)$$

For an finite, irreducible and homogeneous 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:

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

This can be expressed in matrix vector form (in terms of the vector $\pi$ with elements $\{\pi_1, \pi_2, \ldots, \pi_N\}$ and the matrix $Q$ defined above) as $\pi Q = 0$.

We define $p_{ij}$ to be the probability that $j$ is the next state to be entered after state $i$.

**2.2.3 Passage Time Density Calculations for GSPNs**

For a GSPN where $V \neq \emptyset$, we define the passage time from a single source marking $i$ to a non-empty set of target markings $j$:

$$T_{ij} = \inf\{u > 0 : N(u) \geq M_{ij}\}$$

---

3The reachability graph is a CTMC if the set of vanishing states is empty, i.e. the CTMC has no states with zero sojourn time.
where \( M_{ij} = \min\{m \in \mathbb{Z}^+ : \chi_m \in \vec{j} | \chi_0 = i\} \); here \( \chi_i \) is the state of the system after the \( i^{th} \) transition firing \([15]\).

To find this passage time we must convolve the state sojourn time densities for all paths from \( i \) to \( j \in \vec{j} \). We exploit the convolution property of the Laplace transform, this states that the convolution of two functions is equal to the product of their Laplace transforms. We perform a first-step analysis to find the Laplace transform of the relevant density; that is, we first find the probability density of moving from state \( i \) to its set of direct successor states \( \vec{k} \) and then convolve it with the probability density of moving from \( k \) to the set of target states \( j \). Vanishing markings have a sojourn time density of 0, with probability 1, which results in their Laplace transform equaling 1 for all values of \( s \).

If \( L_{ij}(s) \) is the Laplace transform of the density function \( f_{ij}(t) \) of the passage time variable \( T_{ij} \), then we can express this Laplace transform as a system of linear equations given by:

\[
L_{ij}(s) = \begin{cases} 
\sum_{k \notin \vec{j}} q_{ik} L_{kj}(s) + \sum_{k \in \vec{j}} q_{ik} s - q_{ii} L_{kj}(s) & \text{if } i \in T \\
\sum_{k \notin \vec{j}} p_{ik} L_{kj}(s) + \sum_{k \in \vec{j}} p_{ik} L_{kj}(s) & \text{if } i \in V 
\end{cases} \tag{2.6}
\]

If we wish to calculate the passage time from multiple source states, denoted by the vector \( \vec{i} \), the Laplace transform of the passage time density is given by:

\[
L_{\vec{i}j}(s) = \sum_{k \in \vec{i}} \alpha_k L_{kj}(s)
\]

where \( \alpha_k \) is the steady-state probability that the SMP is in state \( k \) at the starting instant of the passage.

Now that we have the Laplace transform of the passage time, we must invert it to get the distribution in the real domain. To do this we use the Laguerre method, which makes use of the Laguerre series representation of \( f(t) \) \([6]\):

\[
f(t) = \sum_{n=0}^{\infty} q_n l_n(t) \quad : t \geq 0
\]

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:

\[
q_n = \frac{1}{2\pi r^n} \int_0^{2\pi} Q(re^{iu}) e^{-inu} du \tag{2.7}
\]

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

The integral in the calculation of Eq. 2.7 can be approximated numerically using the trapezoidal rule, giving:

\[
q_n \approx \frac{1}{2n\pi r^n} \left( Q(r) + (-1)^n Q(-r) + 2 \sum_{j=1}^{n-1} (-1)^j \Re \left( Q(re^{\pi ji/n}) \right) \right) \tag{2.8}
\]
As described in [26], 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 [6]. Slow convergence of the $q_n$ coefficients can often be improved by exponential dampening and scaling using two real parameters $\sigma$ and $b$ [37]. Suitable values for these parameters can be automatically determined using the algorithm described in [26].

Each $q_n$ coefficient is computed as in Eq. 2.8, 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 $Q(z)$.

A advantage of the way in which response time densities are calculated on GSPN models in this implementation is that, as outlined in [21], the technique places no restrictions on the models that can be analyzed. This means using high level specification techniques such as GSPNs one can identify multiple start and target states, including vanishing states (states a with Sorjourn time of zero). This is an advantage over other numerical methods which would, in there calculations, remove vanishing states for faster calculation.

More insight into the approach for the passage time analysis of GSPNs is provided in [21, 26].

### 2.2.4 Moments Calculation for GSPNs

As well as calculating passage-time densities, we can also compute the moments of such densities. The $n^{th}$ (raw) moment $M_{ij}(n)$ of the passage time density from state $i$ into a vector of target states $j$ $L_{ij}(s)$ is obtained by differentiating $L_{ij}(s)$ $n$ times and evaluating the resulting expression at $s = 0$:

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

Moments in GSPNs can therefore be calculated by repeated differentiation of Eq. 2.6 and evaluation of the result at $s = 0$:

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

### 2.3 Performance Trees

#### 2.3.1 Introduction

Performance Trees are a recently proposed formalism for the representation of performance-related queries. They combine the ability to specify performance
requirements, queries aiming to determine whether particular properties hold on system models, and to extract performance measures, quantifiable performance metrics of interest.

2.3.2 Motivation

Expressive power is what sets Performance Trees apart from other performance specification mechanisms, such as stochastic logics, \[8–12, 14, 27, 28\] which in terms of expressiveness are mostly constrained to specific types of queries. Another important characteristic that distinguishes Performance Trees from other formalisms is their ability to represent performance queries in a graphical format, increasing usability significantly, especially for laymen in the area of performance specification.

2.3.3 Operations Available

A Performance Tree query is represented as a tree structure, consisting of nodes and interconnecting arcs. Nodes can have two kinds of roles within queries: operation nodes represent performance-related concepts, such as the calculation of a passage time density for instance, while value nodes represent the inputs to these operations. Value nodes identify attributes such as a set of states, a function on a set of states, an action, or simply numerical or boolean constants. Operation nodes can be interpreted like functions in a programming language, which perform some operation on the supplied inputs in order to obtain a result that is provided as output. In this way, complex queries can be easily constructed from basic concepts by connecting nodes together. Table 2.1 provides an overview of the currently available operation nodes.

Performance Trees are an extensible formalism in the sense that every operation node encapsulates a single self-contained concept, and hence new nodes can be progressively added. Evaluation support for new operations can be provided if it is integrated into our analysis architecture. This extensibility was demonstrated during the course of this project through the introduction of a new operator to the suite of Performance Trees, the Percentile node. The formalism also supports macros, which allow new concepts to be created with the use of existing operators under the cloak of a single Operation node. This is helpful in specifying complex queries without them becoming obtuse.

Performance Trees can be used with many different modelling formalisms, enabled by an abstract state specification mechanism, (details of which can be found in [36]). This report considers Generalised Stochastic Petri nets (GSPNs) as the modelling formalism and presents a GSPN-based application case study.

For an example of the graphical representation of a Performance Tree see Figure 3.3

2.4 Performance Measures and Performance Requirements

In Section 1.1 some of the motivating factors behind the creation of Performance Trees were presented. This Section discusses the state of the art in the field of probabilistic quality property specification. The argument is made for Performance Trees as a
2.4. Performance Measures and Performance Requirements

The overall result of a performance query.

Concurrent evaluation of multiple independent queries.

Passage time density, calculated from a given set of start and target states.

Passage time distribution that is obtained from a passage time density.

Percentile of a passage time density or distribution.

Convolution of two passage time densities.

Probability with which a passage takes place in a certain amount of time.

Transient probability of the system being in a given set of states at a given instant in time.

Raw moment of a passage time density or distribution.

Mean occurrence of an action / firing rate of a transition.

Steady-state probability distribution for a given set of states.

Set of states that have a certain steady-state probability.

Set of states that the system can occupy at a given time.

Boolean operator that determines whether a numerical value is within an interval or possibly within multiple intervals.

Macro, i.e. a new concept created from existing operators.

Boolean operator that determines whether a set is included in or corresponds to another set.

Boolean disjunction or conjunction of two logical expressions.

Boolean negation of a logical expression.

Arithmetic comparison of two numerical values.

Arithmetic operation on two numerical values.

### Table 2.1: Description of Performance Tree operation nodes

Probabilistic verification is distinct from the traditional verification techniques including temporal and real time temporal logics in that they do not seek to guarantee absolute validity of systems. For illustration purposes consider the following different questions one may wish to ask of their system.

Temporal logics seek to assess the questions of the following structure:

- “Will this hazardous event ever occur in our system?”
While probabilistic verification techniques seek to assess questions such as:

- Firstly “With what probability will this hazardous event occur in our system?”
- Secondly “Is this value less than some threshold of occurrence which we would consider acceptable?”

For this reason, probabilistic verification techniques are becoming more prevalent in use as they provide tangible answers to questions such as those above. Rather than receiving strictly binary yes/no responses, system engineers can then assess results and decide upon their acceptability. This does not necessarily mean that traditional verification techniques are obsolete, in-fact many critical systems still need absolute verification to guarantee their safety. The development of probabilistic techniques provides another set of tools that the system engineer has at their disposal in the process of designing and implementing systems.

In [24] a wide survey of 56 academic papers was made and of the 152 distinct probabilistic specification formulae assessed 8 patterns were identified. This research gives an insight to the different kinds of questions a system engineer may wish to ask. Of the 8 patterns the 4 most prevalent patterns accounted for \( \approx 93\% \) of all formulae surveyed. The most prevalent pattern of the 8 accounted for \( \approx 38\% \) of all formulae surveyed. This indicates that systems engineers from many differing fields seek answers to questions of a similar form.

Given the above arguments it is apparent that a formalism which can support just the 4 most frequent patterns is desirable and would serve the vast majority of users. However, for completeness and the convenience of the systems engineer it would be greatly advantageous to develop a formalism which can support all 8 and potentially more patterns should they arise. This would mean that in formulating the probabilistic specification of a system, engineers would only need to work with one formalism, rather than using a different formalism for the many different patterns of query which they may wish to ask of a system. This, as discussed in Section 1.1, is currently the case with performance measures and performance requirements. PTSs can express both these properties and so a tool which provided support for the full suite of performance tree operator nodes would be of value to Systems engineers.

## 2.5 Stochastic Logics

This Section introduces related performance specification formalisms. It also serves to give an overview of the types of performance query that these different formalisms address and also those which they do not.

### 2.5.1 CSL

The most commonly used performance-enabled logic is *Continuous Stochastic Logic (CSL)* [8, 9, 11, 12], which can be considered to provide the framework for all other extended stochastic logic formalisms. CSL operates on continuous-time Markov chains on the state level. Performance requirements are expressed as formulae, which can be of two types. State formulae are true or false in a specific state, while path formulae are true or false along a specific path of the underlying model. The logic has the power to
2.5. Stochastic Logics

express steady-state, path-based and nested constraints. The syntax for these constructs is as follows:

\[
\sigma \triangleq tt \mid a \mid \neg \sigma \mid \sigma \land \sigma \mid S_{\text{vop}}(\sigma) \mid P_{\text{vop}}(\varphi)
\]

\[
\varphi \triangleq \lambda I \sigma \mid \sigma U I \sigma
\]

2.5.2 aCSL

aCSL [28] is an action-oriented variant of CSL. This logic enables the reasoning about system behaviours on a state-based model, enhanced with relevant action information. State formulae are defined just like the \(\sigma\) formulae in CSL, however, aCSL augments path formulae with the following:

\[
\varphi \triangleq \sigma \land U^{<t} \sigma \mid \sigma U^{<t} \sigma
\]

2.5.3 eCSL

eCSL [14] is defined over a higher level of models, in contrast to other logical formalisms, eCSL operates on the model level, rather than at the state level. It was designed express a greater number of performance requirements, including constraints on transient state distributions. eCSL does not support compound formulae and introduces separate layers for the specification of sets of states and of performance criteria. Its syntax is as follows:

\[
\sigma \triangleq tt \mid \neg \sigma \mid \sigma \land \sigma \mid p[N]
\]

\[
\varphi \triangleq \neg \varphi \mid \varphi \land \varphi \mid S_p(\sigma) \mid T_p(\sigma, \sigma) \mid P_p(\sigma, \sigma)
\]

2.5.4 CSL compared to Performance Trees

The most frequently occurring specification pattern highlighted in [24], termed Probabilistic Existence, is of the form

- “A given set of states \(S\) will be entered inside time range \(T\) with probability \(P\)”
- An example from telecoms “In a mobile messaging system, 95% of messages should be delivered in under 30 seconds.”

Both CSL and Performance Trees can formulate queries such as this. The abstract state specification of performance trees allows queries to be composed where multiple start and target states are used, to verify the passage time constraint. Performance Trees use weighted averages over the groups of states specified. While CSL can also represent these types of queries state-by-state verification of passage time constraints is performed. To support multiple state specification in CSL it is necessary to insert additional states into the model’s underlying Markov Chain (see [36] Section IV). This demonstrates the abstract state specification mechanism of PTs simplifies query composition.

In [36] a comparison between queries in CSL and Performance Trees was made. One PT query example which could not be expressed in CSL was provided. This demonstrated
some of the inadequacies of CSL, in this case the inability to extract measures involving high moments of passage time. Although Probabilistic specifications involving higher moments of passage time were not addressed in [24]. One can conceive of a situation where a performance specification such as this would be a desirable system property.

The variants of CSL discussed in this Chapter exhibit differing virtues, eCSL for example can specify constraints on Transient distributions where as aCSL and CSL cannot. Transient queries are of the form

- At exactly time \( t \) what is the probability of the system being in a given set of states?
- For Example, “10 milliseconds after depressing the break peddle what is the probability of left and right breaks both being applied?”

Transient state probability is 1 of the 8 patterns highlighted in [24], these then are questions engineers wish to ask of there systems but which they currently cannot with aCSL or CSL. Performance Trees have the ability to express these kinds of queries, see table 2.1

One further advantage of Performance Trees over CSL is that a CSL formula is usually analyzed by a model checker which returns a true/false answer. Performance Trees are evaluated by a query interpreter that can return various kinds of results, based on which operation nodes are used even sequences of results can be returned.

2.6 Existing Evaluation Tools

This Section provides a brief discussion of other Performance Analysis tools that are currently available. The architecture implemented in this project is evaluated with respect to these tools in Section 6.1.

Möbius

The Möbius [20] software tool is designed for studying the reliability and performance of system models. It allows for the specification of models in varying formalisms, allowing the performance engineer to use the formalism best suited to their task. Multiple modelling languages are supported, these include include: Stochastic Petri nets, Continuous Time Markov Chains and Stochastic Process Algebras. Users can specify measurements at specific time points, over periods of time, or when systems have reached steady state; analogous to the passage time and steady state analysis of PIPE.

PRISM

The PRISM [33] software tool supports probabilistic analysis of systems, specified using Discrete Time Markov Chains, Continuous Time Markov Chains or Stochastic Process Algebras in the form of PEPAs. Models are described using a tool specific language, PRISM; performance characteristics can be be expressed as CSL formulae.

PRISM is supported by a Grid computer, The Midlands eScience Cluster, which performs the computationally intensive calculations required during analysis. Files are submitted to the remote cluster for evaluation and users are able to monitor job progress through a
dedicated monitoring component. Once a job completes remotely, the results are transferred back to the user.

PRISM is a predominantly GUI based tool, although there is command line support. Client facing modules and middleware are implemented using Java™, while code executing on the cluster is implemented in C++.
Chapter 3

Architecture

This Chapter describes the architecture implemented to provide tool support for performance trees. The pre-existing architecture in place at the beginning of this project is outlined in Section 3.1 and details of how it was adapted during the course of this project are provided in Section 3.2. The software components worked with during the course of this project are discussed in Sections 3.3 and 3.4.

The term “component” can be ambiguous in the context of software engineering. For this reason we use the following definition to provide clarity:

A software component is a system element offering a predefined service or event, and able to communicate with other components.

To provide a clearer impression of the components in which work took place for this project the following Key can be used when reviewing Figures 3.1 and 3.2:

- **Blue** indicates completely new components introduced during the course of this project. These include the Performance Tree Evaluation Module, Result Display Module, Performance Tree Division and Dependency Analysis, Analysis Thread Initialisation, PROBI, CONE, PERC and Trivial Node analysis.

- **Yellow** indicates existing components which were adapted during the course of this project. These include: the Performance Tree Editor Module; Data Submission and Results Reception; Data Reception and Results Submission; Result Data Processing; Translation of Model and Subtree to tool specific Language; MOMA; SMARTA; HYDRA.

- **Green** indicates existing components which were left largely unchanged over the course of the project but are still utilized by the new architecture. These include the GSPN Editor Module, DNAMCA, DRMAA, SGE Job Management System and Hardware.

For the rest of this report: the old architecture refers to the architecture which existed prior to the start of this project, the new architecture refers to the architecture which has been implemented in this during the course of this project.
3.1 Pre-existing Performance Analysis Architecture

The performance analysis architecture conforms to the client-server software architecture model. PIPE, the client-side system, initiates a communication session with the Analysis Server on the server-side, which waits for requests from clients. Queries are then evaluated by the Analysis Tools on the Camelot cluster, See Figure 3.1 for a diagram of the old architecture.

![Diagram of Old Evaluation Architecture](image)

**Figure 3.1: Old Evaluation Architecture**
3. Architecture

3.2 New Architecture

A typical performance analysis session proceeds as follows:

1. **Client-Side:** The user specifies a Petri net model in the *PIPE GSPN Editor module*.

2. **Client-Side:** The user specifies a query on the model, this is an abstract representation of the performance measure of interest. Two tools are supported the *Steady State Analyser* and the *Passage Time Analyser*.

3. **Client-Side:** The user sends the model and query data to the server-side and waits for the results.

4. **Server-Side:** Upon receipt of this data, the model and query are translated to a tool specific language by the *Analysis Server*.

5. **Server-Side:** The tool specific representation is sent to the relevant *Analysis Tools* and the job is submitted for execution on the *Camelot Cluster* through the *DRMAA* interface. The job is distributed among the available *Nodes*.

6. **Server-Side:** The *Analysis Server* waits for the completion of the job. Upon which it obtains the results, which have been written to the file system by the *Analysis Tools*, and sends them to the *Client-Side*.

7. **Client-Side:** *PIPE* receives the results, interprets and displays them to the user in the *Steady State Analyser* or the *Passage Time Analyser*.

Although adaptations have been made to the old architecture to support performance tree analysis, the functionality discussed in this Section is still supported by the analysis environment and operates in much the same way as it did before this project was undertaken. This is largely due to the modular nature of the components in the system which ensured that the performance tree analysis could be kept separate from the existing tools allowing for backwards compatibility.

### 3.2 New Architecture

The architecture implemented for the evaluation of Performance Tree queries on GSPN models is shown in Figure 3.2.

A typical Performance Tree evaluation session proceeds as follows:

1. **Client-Side:** The user specifies a Petri net model in the *PIPE GSPN editor module*.

2. **Client-Side:** The user specifies a Performance Tree query in the *Performance Query Editor*.

3. **Client-Side:** The user begins an evaluation session by sending the model and query data to the server.

4. **Client-Side:** The *Result Display Module* displays an evaluation window which details the progress of the query evaluation, as results become available the user is informed and the results are displayed (see Figures 3.4 and 5.3(a) for an example of this window).
3.2. New Architecture

3. Architecture

Figure 3.2: Performance Tree Evaluation Architecture
5. **Server-Side:** The Analysis Server, upon receipt of this data, evaluates the query, breaks the query into individual tasks, specifying the order in which evaluation must take place if dependencies exist.

6. **Server-Side:** The individual tasks are submitted for evaluation concurrently each task is allocated its own evaluation thread, which initiates a session with Analysis Tools and wait for its completion to obtain results.

7. **Server-Side:**

   Simpler tasks, such as arithmetic comparisons or arithmetic operations, are evaluated by a Java\textsuperscript{TM} based tool, the Trivial Analyser.

   Tasks requiring higher performance are supported by tools written in C++ and execute on the Camelot Cluster.

8. **Server-Side:** The Analysis Server waits for the completion of the individual tasks, results from the Analysis Tools as acquired from the File System and Result data Processing occurs, results are sent to client as they become available.

9. **Client-Side:** The Result module receives results for each operation node of the Performance Tree, displaying them in the Result Display Module when results for all nodes are received evaluation is complete.

10. **Client-Side:** The client can now evaluate results from individual nodes as well as the overall result of the query in the Result Display Module.

### 3.3 Client-Side Application

The PIPE Platform-Independent Petri net Editor \[1\], is a Java\textsuperscript{TM} based tool for the specification of Petri net models and the extraction of the performance characteristics from them. PIPE is equipped with pluggable analysis modules which perform tasks such as invariant analysis, steady state analysis and passage time analysis. The Performance Tree evaluation module was added to PIPE as a new analysis module in the course of this project.

#### 3.3.1 GSPN Editor Module

The GSPN Editor Module allows for the construction of Petri net models on a canvas using tools that enable the placing and connection of place and transitions. This Module pre-existed this project and was largely untouched during the projects implementation. See Figure 5.1 for an example Petri net composed in the editor.

#### 3.3.2 Performance Query Editor Module

The Performance Query Editor module, which also pre-existed this project, implements a graphical interface for the composition of Performance Tree queries. These are constructed on a canvas using a toolbar that enables the placing, connection and manipulation of operation and value nodes. As a performance tree is composed an object graph constituting nodes and arcs is built, when the performance tree is sent for
evaluation this structure is serialized and sent to the Server-Side. Before a query is to be submitted to the Analysis Server, validation is performed. This prevents users from building illegal or incomplete performance queries, for instance by connecting incompatible operation nodes (causing a type violation) or by not supplying required arguments to operation nodes.

Figure 3.3: Performance Query Editor module showing a performance requirement verification query

The query creation process is further simplified by an automatic query interpretation mechanism that translates the Performance Tree query under construction into its natural language equivalent. This is particularly helpful to those users of the tool who have not yet developed familiarity with Performance Trees as it provides them with the means to intuitively verify their queries. Figure 3.3 depicts a performance requirement verification query being built with the query designer interface. It also shows how a query is represented in natural language while being constructed.

3.3.3 Performance Tree Evaluation Module

When a GSPN model of a system and an applicable Performance Tree query have been specified, PIPE uses Java object serialization to send the GSPN model and the Performance Tree query via a socket connection to the Analysis Server. Upon initialisation of a socket connection with the Analysis Server, a dedicated thread is made available to receive the incoming data and to commence the evaluation process. The model and query data are sent to the server-side along with some application configuration settings such as the number of processors to distribute the job across on the Camelot Cluster.
Upon submitting the Performance Tree and model for evaluation a GUI window displaying the query appears, this provides continuous feedback on the progress of the query evaluation on the Analysis Server through the use of a traffic light system. Before analysis of a node has begun it is coloured **red**; once analysis begins the node flashes **amber**; once analysis of a node is complete it turns **green**. At this point the results for that node have been received, the user can access individual results by clicking on the node. Results are sent from the server to client as they become available, so before the evaluation of the entire query is complete users can still evaluate interim results. This system allows users to understand how the execution of different nodes in their query is progressing (see Figure 3.4). The user is notified that the evaluation of the entire query is complete when a progress bar fills and turns green at this point all results are available and so all nodes are coloured green, this is shown in Figure 5.3(a) in Chapter 5.

![Figure 3.4: Evaluation window displaying PT with PTD and Percentile nodes, yellow indicators show evaluation is in progress.](image)

### 3.4 Server-Side Application

#### 3.4.1 Analysis Server

The analysis server acts as middleware between PIPE and the analysis tools. It performs the model and query translation, submits analysis jobs to the camelot cluster for evaluation and returns the results to the client-side system.

The Analysis Server is a java based tool, its main responsibility is the inspection of performance queries and the appropriate distribution of work among the dedicated analysis tools hosted on the Grid cluster. Once the server has obtained serialized objects
representing the system model and the performance query, analysis of the query is performed and an evaluation order is established based on the dependencies between nodes to ensure evaluation can proceed successfully, for more discussion of this issue see Section 4.1.

Depending on the Operation nodes used in the query several different analysis tools could be required to be used, the relevant parts of the query need to be translated to a tool specific language. To do this firstly, a translation to the DNAmaca [31] model specification language is be performed, which describes the structure of the model, this language is used by all analysis tools. Subsequently, the analysis tools are invoked by the analysis server, parsing the model and relevant parts of the query data into C++ code compiling the output with a probabilistic hash-based state space generator library. This is linked with a pre-compiled performance analysis library and executed in order to compute the requested performance measures. Once the server has distributed the jobs amongst the analysis tools, it awaits the results of the calculations, which it are then sent to the client-side.

### 3.4.2 Analysis Tools

As discussed in Section 3.1, tools were already available for the calculation of steady-state measures, first passage time and transient distributions, as well as higher-order moments based on the underlying Markov chain of the GSPN model [22, 31]. The integration of these tools into the analysis pipeline enables the evaluation of the majority of Performance Tree operator nodes. In the course of this project additional tools which work in conjunction with the above have been developed to provide support for remaining operators.

**DNAmaca**

DNAmaca [31] is a Markov chain steady-state analyser that can solve models with up to $O(10^7)$ states. It features model and performance measure specification in its input language, functional and steady-state analysis and the computation of performance statistics, such as the mean, variance and standard deviation of expressions computed on states in the model. The raw distribution from which the above performance statistics are calculated can also be obtained. DNAmaca is used in the new architecture for the evaluation of the SS:P and FR nodes.

**Smarta**

SMARTA [22], the Semi-MArkov Response Time Analyser, is a parallel MPI-based analysis pipeline for the iterative numerical analysis of passage time measures in very large semi-Markov models (including GSPNs). This is done by first calculating the Laplace transform of the passage time between the given start and target states, an inversion method, such as Laguerre or Euler, to obtain the distribution in the real domain. This tool is used to evaluate the PTD and Dist nodes, and also works in conjunction with Cone, Probl and Perc to provide support for Convolutions, ProbInInterval and Percentile nodes.
Moma

MOMA is a performance analyser that calculates \( n^{th} \) order (raw) moments using a Laplace transform-based method. It is used to evaluate the Moment node.

Hydra

Hydra [22], the HYpergraph-based Distributed Response time Analyser, is a performance analyser that performs Transient analysis. It is used to provide support for the Prob In States node. Although a short coming of this tool is that it cannot evaluate distributions with start or target states with vanishing states (states with a sojourn time of zero).

ProbI

ProbI is a tool which works with SMARTA to provide support for the Prob in Interval node.

Cone

Cone is a Convolution tool which works with SMARTA to provide support for the Convolution node.

Perc

Perc is a tool which works with SMARTA to provide support for the Percentile node.

3.4.3 Cache verification

For enhanced speed and efficiency, the Analysis Server incorporates a disk-based cache mechanism that stores the results of performance queries evaluated by the analysis tools. In order to differentiate between different queries on the same model, a hash of the model description and the performance query specification is performed separately, using a hashing algorithm with a very low probability of clashes. This is used to create a two-level structure in which the computed performance measures can be stored. Before any computation takes place, a cache look-up is be performed for the given model. If a match is found, the hash of the current query is compared to all existing hashes of queries on that particular model in the cache. If a further match is found, the query has already been evaluated on the model and the results are already available. If not, the query has to be evaluated, with the results of this being stored in the cache.

For DNAmaca, each performance measure is hashed separately and the corresponding cache location contains the computed steady-state result (either a state measure or count measure). For SMARTA and MOMA, the cache is be particularly useful as calculating the density or distribution of a passage time requires the solution of at least 400 sets of linear equations of the form shown in Eq. 2.6. Recalling Eq. 2.8, the values at which the Laplace transform of the passage time must be evaluated are independent of the range of \( t \)-values at which the final answer is required. Thus, for a given model and
set of initial and target states, results can be computed at any value of $t$ using the same values of the Laplace transform of the passage time. We therefore store the computed values of this Laplace transform indexed by the value of $s$ to which they correspond.

### 3.4.4 Camelot Cluster

The Camelot cluster is where the analysis tools are executed. It is the computational cluster forming the backbone of the evaluation engine, it consists of 16 dual-processor dual-core nodes, each of which is a Sun Fire x4100 with two 64-bit Opteron 275 processors and 8GB of RAM. Nodes are connected with both Gigabit Ethernet and Infiniband interfaces; the Infiniband fabric runs at 2.5Gbit/s, managed by a Silverstorm 9024 switch. Job submission is handled by Sun Grid Engine, a middleware that configures and exposes the cluster as a computational Grid resource. Clients submit sequential and parallel (MPI) jobs to Grid Engine via the Distributed Resource Management Application API (DRMAA).
Chapter 4

Implementation

This Chapter describes how we went about adding the components discussed in Chapter 3 to the existing Analysis Architecture. We briefly describe how new components were integrated into the existing PIPE and the Analysis Server tools. Following that we lay out some of the challenges this presented, how they were addressed and overcome.

4.1 Evaluation of performance trees with dependent nodes

The enhanced expressibility and flexibility of Performance Trees provides a rich suite of operators to systems engineers with which they can simply express the performance characteristics they wish to extract from a model. Unfortunately, it is exactly this characteristic which presents difficulties in implementing a system to evaluate them. A single Performance Tree query often requires the calculation of many potentially dependent measures, this gives rise to an ordering in which results must be evaluated. Therefore, the scheduling of node evaluation requires an understanding of the dependencies of the sub-trees of the current node.

Consider Figure 4.1 with Operation Nodes $X$, $Y$ and $Z$ in a large performance tree.

Figure 4.1: Example 1 Performance Tree demonstrating node dependencies
When evaluating this performance tree it is clear that $Y$ must be evaluated before $X$ can be as $X$ requires $Y$’s result to be used as input; it does not matter when $Z$ is evaluated with respect the $X$ or $Y$ as it has no affect on the evaluation of these nodes.

A trivial approach to evaluating trees such as this would be to perform a bottom-up in-order traversal of the tree evaluating one node at a time, however, this would be highly inefficient. A more efficient approach would be to realise that the evaluation of independent nodes\(^1\) can take place in parallel, so that multiple elements in the tree can be submitted for evaluation to the cluster at the same time. Dependent nodes are then queued and scheduled for evaluation once their dependents have been evaluated. This gives rise to an *execution hierarchy* with distinct levels which provide the order that sets of Nodes can be evaluated in. Each set waits for the completion of evaluation of all nodes in the set before it.

### 4.1.1 The Problem

This approach provides an acceptable order in which trees can be evaluated and allows for some parallelization of node evaluation, however it still possesses some short comings.

Consider Figure 4.2 with Operation Nodes $X$, $Y$, $Z$ and $K$ in a large performance tree. We will have 2 distinct sets for our execution hierarchy. Set 1 : \( \{ Y, Z \} \), Set 2 : \( \{ X, K \} \). Both $X$ and $K$ must wait for $Y$ and $Z$ to complete evaluation but they only really need to wait for their respective child node.

As we cannot know before run time how long $Y$ or $Z$ will take to complete evaluation the only way to approach this problem is to use some sort of notification system. When a given node’s evaluation is complete it informs our *Analysis Server* and can then send parents for evaluation knowing they have all the input they require. For example when $Y$ is complete the *Analysis Server* is notified and it can send $X$ for evaluation. In this way parallelisation can be increased, and this is exactly the approach implemented in this project.

With this refined approach we have all the requisite knowledge to approach our problem and can begin to formulate an algorithm.

---

\(^1\)Nodes that do not appear in each others subtree, i.e. the nodes exist on different branches of the PT
4. Implementation 4.1. Evaluation of performance trees with dependent nodes

1. Break the performance tree down into single Operation Nodes which only have knowledge of the nodes in the tree that exist below them, their subtree.

2. Iterate through each operation node’s subtree, group operation nodes by the number of unevaluated nodes they have in their subtree. This constructs our execution hierarchy.

3. Schedule each node for evaluation adhering to the execution hierarchy, those nodes which have unevaluated nodes in their subtree must wait to be notified that they can proceed by their direct child node before starting evaluation, in this way they can be sure they have the input they require to proceed.

4. Evaluation proceeds up the tree until all nodes have been evaluated and the overall result of the query is obtained.

Many further optimizations to this scheme could have been implemented; some of these are discussed in Chapter 7 on page 59.

4.1.2 Multi-Threaded Evaluation and Java™

After some investigation it became clear that implementing a system providing notification upon the completion of node evaluation from scratch would be quite time consuming. Also due to the inherent non-determinism of these types of systems they are notoriously difficult to debug! Thankfully after some reading up on some new Java™ features [30] we discovered a package introduced in Java™ 5.0 which would provide the notification mechanism we needed, this could be implemented in such a way that much of the actual notification could be handled directly by Java™ and the structure of the Performance Tree would define the evaluation schedule as tasks complete in an ad-hoc fashion.

Java.util.concurrent package [2] provides a set of utility classes for concurrent programming. By implementing some of the interfaces of this package we would have the requisite framework to solve this problem. The members of the package we used are described in Sections 4.1.3, 4.1.4, 4.1.5 and 4.1.6.

4.1.3 Callable interface

In Java™ the Runnable interface encapsulates an asynchronous task, a method taking no arguments and with no return type. It is typically implemented by classes which need to run on a their own thread but make use of inheritance, have a super-type, and so cannot simply extend the Thread class (Java™ does not support multiple inheritance), they must therefore implement this interface.

The Callable interface is very similar to Runnable except that it can return a value. Callable is a generic type, this means it can be implemented to return an object of any type, for example a Callable < Integer > returns an Integer result and a Callable < Giraffe > would return an object of type Giraffe. The implementors of the Callable interface can execute an asynchronous task and return an object when they complete. Another nice feature is that the Callable interface allows checked Exceptions to be thrown from their Call method (analogous to the run method of Runnable), this would allow us to recognise and diagnose problems in analyzing particular nodes.
4.1. Evaluation of performance trees with dependent nodes

By implementing the Callable interface we would have a mechanism to run an asynchronous task, the evaluation of a node, and get a value returned to us when this evaluation completed.

4.1.4 Future interface

A Future object represents the result of an asynchronous task. As the name implies this allows one to start a computation and pass the Future object to someone who will need it in the future, the owner of the result can then obtain the result when it’s ready.

Future objects also make use of generics, so as before a Future < Giraffe > will allow the owner of this object to get access to the Giraffe object when an asynchronous task completes. If the owner attempts to get the Giraffe, via a getter method, before the asynchronous task is completes this request blocks until the computation is complete.

The way in which we use the Future in evaluation of nodes is to start evaluating all nodes which can be evaluated straight away and then pass the Future object we obtain to their parents. In this way the parent nodes naturally wait until the results they need are ready ensuring if we begin tasks adhering to the execution hierarchy then we do not violate any of the dependencies.

4.1.5 ThreadPool class

Creating many short lived threads is an expensive interaction as it requires the Java<sup>TM</sup> virtual machine to interact with the operating system. Furthermore, creating a large number of threads and running them concurrently greatly degrades the performance and can even crash the Java<sup>TM</sup> virtual machine (VM).

For these reasons one should use a ThreadPool, a set of idle threads ready to run tasks submitted to them. This prevents expensive creation of many threads as threads in the pool are reused and the size of the pool can be bounded so we do not affect the performance of the Virtual Machine. Another benefit of a ThreadPool is that it allows you control over the entire pool of threads running, one can easily check if all tasks have finished executing, or one can cancel all tasks in the pool at once.

The ThreadPool class would allow us to manage the many threads which may be run to evaluate an entire performance tree query. With the potential for many concurrent clients evaluating performance trees this is important because, as discussed, if too many threads were created and ran concurrently then we may experience problems with the Virtual Machine.

4.1.6 ExecutorService interface

Using an ExecutorService allows asynchronous, Callable or Runnable, tasks to be submitted for execution in a ThreadPool returning a Future object of the generic type of the return type of the Callable submitted for execution. If one were to submit a Callable < Giraffe > for execution through an ExecutorService then the submitter would be returned a Future < Giraffe > object. When the owner of this object used the getter to attempt the obtain the Giraffe object the thread executing the method would be caused to wait until the task completed.
4. Implementation

4.1. Evaluation of performance trees with dependent nodes

**ExecutorService** provides us with a clean mechanism to “fire and forget” the nodes we need to evaluate. We can start an evaluation, pass the **Future** object around and let the structure of the performance tree manage the actual scheduling itself by starting the node evaluation off adhering to the *execution hierarchy*, then when nodes we cannot evaluate yet attempt to get input from their children through the **Future** interface they have to wait until these results are ready to be used.

### 4.1.7 Supporting Classes

Figure 4.3 provides an overview the key classes in the Analysis Server and shows the general structure of this component. This section provides a brief description of some of these key classes.

**Figure 4.3:** Class Diagram of the Analysis Server and related components

---

**ServerAction**

The **ServerAction** class is the main co-ordinating class that manages an evaluation session. When a client initialises a session this class is created and spawned as a thread.
4.1. Evaluation of performance trees with dependent nodes

that will handle the rest of the evaluation session. This class is responsible for the creation of the following classes:

- **SubtreeCollection**, a list holding Subtrees in the execution hierarchy.
- **TranslateQueryTree**, this class translates subtrees to a tool specific representation (as detailed in Section 4.2).
- **ThreadPoolExecutor**, a class implementing the ExecutorService.

**AnalysisExecutor**

This class implements the Callable interface, instances of this object are submitted to the ThreadPoolExecutor which allocates an available thread upon which the object executes.

**Subtree**

A subtree is a structure which represents a portion of a Performance Tree. Before analysis is performed the Performance Tree is broken down into subtrees which represent the individual operations which must be performed to evaluate the entire Performance Tree. It constitutes:

- An Operation Node
- Value nodes (of which there are usually two) which are the input to the Operation Node.
- A collection of all the subtrees in the Performance Tree which are descendants of this node.
- A result field of type `Future < ResultWrapper >`

Each subtree represents a single operation to be performed and contains all the data required for it to be evaluated. When an `AnalysisExecutor` is submitted for evaluation the Future object that is received can the be used to obtain the result when evaluation completes. Using the result field of this class results of nodes below the operation node represented in this subtree can be obtained and in this way results are passed up the Performance Tree allowing evaluation to proceed. It should be noted that the number of subtrees which the Performance Tree is broken into is equal to the number of operation nodes in the query.

**ResultWrapper**

The ResultWrapper class is an encapsulation of the result of evaluating an Operation Node. The ResultWrapper class is an abstract super class from which many other result classes derive. Due to the many varied results which different performance tree nodes can return different concrete classes are required. A node which evaluates to a Boolean or Numerical result is very simple to encapsulate with the native types in Java™ (Bool or Double). The evaluation of a Passage Time density node results in a density, a set of probability and time points which are plotted in on graph; clearly to encapsulate this result a different sub-type is required from the Boolean or Numerical results.
Using the ResultWrapper as the return value of the call method implemented from the Callable interface means evaluation can be handled generically. Results are passed up the Performance Tree in a unified way with each concrete ResultWrapper class representing the different result types of the operation nodes. This allows the type of result being returned to vary independently of the way results are passed between subtrees.

4.1.8 Bringing it all together

Now with an understanding of the motivation behind the algorithm and with knowledge of the tools which have been used to implement it we can describe how the evaluation of the Performance Trees is performed.

An evaluation session proceeds as follows (numbers corresponding to those in Figure 4.4).

1. The user sends a model and query for evaluation from PIPE to the ServerAction class.
2. A collection of Subtrees is created
3. The collection is sorted by a simple heuristic, the number of unevaluated subtrees which exist in its descendants collection in ascending order.
4. The ordered list is obtained by ServerAction
4.2 Translation of Performance Tree to Tool Specific Languages

5. An ExecutorService is created by a static factory method

   **Iterate through the ordered list of subtrees.**

6. A callable (task) is created for the subtree

7. The callable is submitted for evaluation and a Future object is obtained.

8. This subtree’s result is set to be Future object.

   **Go to 6 until the end of the subtree list is reached.**

   **Begin (asynchronous) evaluation thread.**


10. The value nodes of the subtree are obtained, forming the input to this operation node.

11. The task is submitted for evaluation to the Analysis Tools.

12. The Analysis Tool evaluation completes.

13. The Callable Task completes and the Result is returned causing waiting threads to continue.

   **End evaluation thread.**

14. The future object is used to obtain the results for this subtree

15. The results of the Performance Tree are obtained by ServerAction class

16. The results are sent to PIPE to be displayed to the user

Points 10-13 represent the execution of a thread, the time this takes to execute accounts for the vast majority of time spent evaluating a performance tree. Submission of subtrees for evaluation is asynchronous and so the subtree loop does not wait for the completion of the Analysis Tools. While separate threads are executing other subtrees are submitted, the order of submission is dictated by the execution hierarchy. A Subtree with unevaluated descendant subtrees waits for their completion at 10 when the get() method on the future object is called. When the Task to which this Future corresponds completes the get method returns and so evaluation can proceed.

Other events which occur during thread execution are the notifications of the evaluation status for the traffic light system and sending interim subtree results to PIPE, these messages have been left out of the diagram for simplicity.

4.2 Translation of Performance Tree to Tool Specific Languages

Prior to tasks being submitted to the Analysis Tools for evaluation to be performed, the structure of the Petri net model is analysed and translated to the to the DNAmaca [31] model specification language, which describes the structure of the model. As tasks are submitted to the Analysis Tools this model file has query related data appended to it.
that specifies the particular performance measure being requested, this makes use of the augmented input language described in [22]. For steady state and passage time measures the input language is as follows:

\[
\text{passage_time_measure} = \text{\textbackslash passage}\{
\text{\textbackslash sourcecondition}\{\text{<boolean expression>}\}
\text{\textbackslash targetcondition}\{\text{<boolean expression>}\}
\text{\textbackslash t_start}\{\text{<real expression>}\}
\text{\textbackslash t_stop}\{\text{<real expression>}\}
\text{\textbackslash t_step}\{\text{<real expression>}\}
\}\n\]

\[
\text{state_measure} = \text{\textbackslash statemeasure}\{\text{identifier}\}\{
\text{\textbackslash estimator}\{\text{(mean | variance | stddev | distribution)}*\}
\text{\textbackslash expression}\{\text{<real expression>}\}
\}\n\]

\[
\text{count_measure} = \text{\textbackslash countmeasure}\{\text{identifier}\}\{
\text{\textbackslash estimator}\{\text{mean}\}
\text{\textbackslash transition}\{\text{all | \{identifier\}*}\}
\text{\textbackslash expression}\{\text{<real expression>}\}
\}\n\]

The value nodes attached to the operation node of the subtree in question are examined to provide: the source and target conditions for passage time measures (PTD nodes); the state function expression for state measures (SS:P nodes); the action expression for count measures (FR nodes).

With the model and query data translated to the tool specific language, a file is written to the file system, the path to this file is provided to the analysis tool and when it is invoked this file is parsed.

### 4.3 Client-Server Network Interaction

To support performance tree evaluation the existing mechanism by which model and query data was submitted to the Analysis Server was changed. The previous single TCP Socket which was used has been replaced by two TCP Sockets with more distinct roles, this reflects the fact that the new architecture requires a greater number of messages to be sent between both client and server in the form of notifications on the status of evaluation and also volumes of data sent have increased as far more results are now returned to the user in the form of data points for graphs, textual outputs etc.

- The first Socket, *The Information Socket*, handles initialisation of the connection between client and server, and subsequent notifications on the state of evaluation.
- The second Socket, *The Data Socket*, sends data between client and server in the form of model and query data and results.

The Java\textsuperscript{TM} Socket implementation can operate with several types of “streams”. As the *Information Socket* is concerned with notifications far simpler data is required to be sent
4.4. Logging System

As the evaluation architecture is implemented to handle requests from many concurrent clients, a logging system was implemented to facilitate quick diagnosis of runtime issues. This system makes use of the **Java.util.logging** package [3], which provides a thread safe mechanism to maintain logs; these detail the state of the Analysis Server and its interactions with other components in the architecture. Another use of this system is that it is helpful in understanding how code executes on the Analysis Server. As this component runs on a remote machine it cannot be debugged to determine the source of errors, this system was invaluable during the development of the Analysis Server to discover the source of bugs.

The **Java.util.logging** package uses **Handlers** to communicate with the outside world, using different types of handlers logs can be communicated in many different ways, a file handler creates file and appends logging output to this file, a socket handler sends logs over a remote connection using a TCP socket, both of these types of handlers are used in the Analysis Server to allow users to view the logs from their evaluation session. A potential extension to this system would be to use logging output to dynamically generate a website that could report on the runtime status of the evaluation architecture, as varying logging output formats are supported, including XML, this would be a relatively simple addition to the system.

The logging system is used as the reporting mechanism for thrown exceptions. As the code runs remotely, a simple console based exception reporting mechanism would be insufficient for our purposes, especially as many users may be using the system at once. For this reason individual sessions are separated and logs are placed on the file system indexed by a two level date-time-stamp to make it simple to find the logs relevant to a particular session. Another useful property of this *Java™* package is that logs of different **Levels** can be used which provide a description of the severity of a particular log entry. This means log messages relating to debugging code can be easily filtered from important messages, for instance indicating exceptions have been thrown.
4. Implementation

4.5 Result Display Module

The Result Display Module is a swing based GUI interface. It makes use of the familiar idea of a tabbed window, allowing users to switch between a group of components by clicking on a tab with a given icon. The initial tab displays the Performance Tree being evaluated (see Figure 5.3(a)), as results become available other tabs are opened, for example to display graphs (see Figure 5.3(b)). This GUI makes use of the ResultWrapper class hierarchy (see Section 4.1.7) using runtime type information to determine how to display results, to facilitate this a Bridge design pattern [23] was implemented. A progress bar is placed at the bottom of the window to indicate how evaluation is proceeding. In general we will not know how long an evaluation session may take to complete, for this reason the progress bar changes to an indeterminate mode which indicates that the session is executing but we do not know how long it may take to complete; when information relating to the progress of the evaluation session is received the progress bar becomes determinate again and progresses. This gives users a general impression of how the evaluation session is proceeding. (See Figure 3.4 for a screenshot of the progress bar in indeterminate mode and Figure 5.3(a) for determinate mode)

4.6 Camelot Cluster Configuration

4.6.1 DRMAA Singleton

The Distributed Resource Management Application API (DRMAA) is used to submit Analysis Tool jobs to be executed on the Camelot cluster. This API provides support for job submission, job monitoring and control, and retrieval of the finished job status. For this reason a cluster such as Camelot is only permitted to have one open session with a DRMAA distributed resource management system. Without this caveat multi DRMAA instances could be used to wreak havoc on the cluster, for example by requesting different control settings.

The old architecture was not designed to support multiple concurrent clients. For this reason when using DRMAA: it opened up a new DRMAA session; submitted a job for execution; waited for the finished job status; closed the session. This implementation was not conducive to the single session requirement of DRMAA as multiple clients would cause attempts to open multiple sessions; when multiple clients attempted to use DRMAA at the same time the request that arrived last was denied as a session was already opened.

This issue presented problems for the new architecture as in a single Performance Tree query many DRMAA sessions could be required to submit jobs for execution. Initially a queue was implemented to to provide access to the API, however this would mean that only a single analysis tool could execute at a particular time and this would not make full utilisation of the Camelot Cluster's computational power.

After consulting some documentation on DRMAA [4] we found a potential solution. While only one DRMAA session is permitted at any one time, that DRMAA session can have multiple submissions made through it. By implementing a Singleton design pattern [23] one session instance could be passed around by the Analysis Server and used to submit analysis tasks. Thankfully as the DRMAA implementation is thread safe this would present no concurrency issues.
4.6.2 Processor allocation

The Camelot Cluster is supported by 16 nodes, each with 4 processors (2 dual cores). When a job is submitted to the cluster for evaluation a number of processors, up to the maximum available, can be requested to be used. Any Analysis tool which makes use of the Message Passing Interface (MPI) [5] can then distribute their task across the number of processors requested, if the task is sufficiently large this can significantly reduce the time to perform evaluation. Currently their is only one of the Analysis Tools that is sufficiently computationally intensive to make using MPI effective, this is SMARTA the passage time analyser.

4.7 Performance Tree Node Evaluation

This Section provides some implementation specific details of the tools used to analyze performance tree nodes.

4.7.1 DNAmaca: Steady state Analyzer

Using DNAmaca, steady-state performance statistics for the GSPN model are derived by generating and then solving a continuous-time Markov chain (see Section 2.2.2). From the Markov chain’s steady-state probability distribution, high-level performance measures such as throughput and mean buffer occupancy can be derived. These correspond to the FR and SS:P Performance tree operators.

4.7.2 SMARTA: Passage Time Analyzer

Passage time analysis on a GSPN model is performed as described in Section 2.2.3. By obtaining a Laplace transform representation of the first passage time distribution and then solving this linear equation for values of the Laplace transform at several points in the function. The Laguerre inversion method is then used to obtain a numerical solution for the time domain equivalent of the transform. If the operation nodes PTD and Dist are part of a Performance Tree query, the user can expect a probability density function (pdf) and a cumulative distribution function (cdf) as a result after the passage time calculation has been completed by SMARTA (see Fig. 3.2).

An interesting issue that arises in displaying the results of the above nodes is the automatic determination of the time range over which a pdf or cdf should be plotted. In both cases, we will do this by establishing at what time value t the cdf approaches 1 within some ǫ bound. The pdf or cdf is then plotted between 0 and t.

We consider the complementary cumulative distribution function (ccdf) \( F^c(t) = 1 - F(t) \), since it has a structure more suited to numerical inversion (since \( F^c(t) \) is a non-negative decreasing function with \( F^c(t) \rightarrow 0 \) as \( t \rightarrow \infty \)). Using Laguerre series representation,

\[
F^c(t) = \sum q'_n l_n(t)
\]  

(4.1)

with \( Q'(z) = q'_n z^n \). It can be shown that

\[
Q'(z) = \frac{-2(1-z)}{(1+z)} (Q(z) - (1-z)^{-1})
\]  

(4.2)
Thus, the Laguerre coefficients of the ccdf $q'_n$ can be computed based on the Laplace transform $f^*(s)$ of the density function. Furthermore, based on the following relationship, the Laguerre coefficients of the pdf $f(t)$ can also be recursively computed from $q'_n$:

$$q_n - q_{n-1} = -\frac{1}{2}(q'_n + q'_{n-1})$$  \hfill (4.3)

Finally, the cdf and pdf are obtained as:

$$F(t) = 1 - \sum q_n l_n(t) \quad \text{and} \quad f(t) = \sum q_n l_n(t)$$  \hfill (4.4)

The time range of interest for the user is given by the time at which $F^c(t) \to \epsilon$ where $\epsilon$ is chosen as an arbitrarily small, positive value. Eqs. 4.2 and 4.3 are described in more detail in [32].

### 4.7.3 MOMA: Moment Analysis

If the Moment operation node is included in a query then MOMA is be invoked to calculate the required moments using the formula in Section 2.2.4.

### 4.7.4 Moments of Steady State probability distributions

While the MOMA tool supports the calculation of Moments on Passage Time Density a tool was required to calculate Moments of Steady State distributions returned from DNAmaca, as the SS:P node also returns a distribution. As these results are discrete the calculation of Moments can be performed by an iterative procedure where the $n^{th}$ Moment of the distribution function $F$ is obtained as:

$$M(n) = \sum (X^n) dF(x)$$  \hfill (4.5)

### 4.7.5 PERC: PERCentile Analysis

PERC is used to obtain a percentile of a passage time distribution, it makes use of work performed by SMARTA in the Passage Time Analyser, SMARTA. SMARTA calculates the distribution points and Laguerre coefficients of the ccdf for use during evaluation of the PTD node. These values are output into separate files and can then be cached and used by PERC to calculate the Percentile which saves computation time and keeps the complexity of tools such as PERC to a minimum. As the Percentile operator is defined to only operate on the PTD node it is a fair assumption that distribution points and Laguerre coefficients will always be available to the tool.

When SMARTA establishes a sensible time range across which to represent a distribution it outputs 600 equidistant points across the time range along with their corresponding probability value, these are used by the Dist operator to be plotted as a graph. As we are dealing with a cumulative distribution the probability range will start at zero and approach one at the end of the time range. PERC uses the fact that distribution points represent a range of “interesting points” to find a good starting point to find the Percentile of interest.
An important property of the Laguerre method is that Laguerre coefficients can be used to invert the distribution at any point across the entire distribution. This is helpful to us as a single set of around two hundred coefficients can then be used to obtain the probability value at any point across the time range. With a good starting point, obtained from the distribution points, we can then “home in” on the percentile value we are looking for using a binary chop search algorithm.

For finding the $P^{th}$ percentile firstly we convert $P$ to a Quantile $Q$ which is the decimal representation of $P$, i.e. $Q = P/100$. The pseudo code is then as follows:

01. for each distribution point pair $T_n, P_n$ {
02. \hspace{1em} if($P_n > Q$) \hspace{1em} \hspace{1em} lowerTimeBound = $T_{n-1}$
03. \hspace{1em} \hspace{1em} upperTimeBound = $T_n$
04. \hspace{1em} \hspace{1em} break
05. \hspace{1em} }
06. \hspace{1em} do { \hspace{1em}
07. \hspace{2em} midTimePoint = (lowerTimeBound + upperTimeBound) / 2
08. \hspace{2em} probability = getProbFromCoefficients(midTimePoint)
09. \hspace{2em} if (probability > $Q$)
10. \hspace{3em} upperbound = midTimePoint
11. \hspace{3em} else lowerbound = midTimePoint
12. \hspace{2em} }
13. \hspace{1em} while(probability - $Q > 10^{-6}$)
14. \hspace{1em} return midTimePoint

The $P^{th}$ percentile is then the value in the midTimePoint variable. As this algorithm converges very quickly, halving the search space in each iteration, PERC converges to a solution in few iterations.

### 4.7.6 PROBI: PROBability in Interval Analysis

PROBI is used to obtain the probability a passage takes place in-between a particular time range. Similarly to PERC, PROBI uses the Laguerre coefficients output from SMARTA to calculate the probability of interest. In the case of PROBI the calculation is far simpler to carry out by virtue of the fact that the Laguerre coefficients are from a cumulative distribution function. This means only two probabilities need to be calculated to obtain the probability the passage take place inside the given time range. $t_1$ is used to calculate the cumulative probability the passage takes place between $0 - t_1$, $t_2$ is used to calculate the cumulative probability the passage takes place between $0 - t_2$ the probability the passage takes place between $t_1$ and $t_2$ can then be calculated as follows:

$$ p = \sum q'_n l_n(t_1) - \sum q'_n l_n(t_2) \quad (4.6) $$

### 4.7.7 CONE: CONvolution of passagE time densities

This tool exploits the convolution property of the Laplace transform, this states that the convolution of two functions is equal to the product of their Laplace transforms. Given
two functions \( f(x) \) and \( g(x) \) and their respective Laguerre coefficients \( f_n \) and \( g_m \), we can calculate the coefficients of convolved function \( (f * g)(x) \) as follows:

\[
(f * g)_n = \sum_{m=0}^{n} f_n^m - m g_m^m
\]  

From this the passage time density is calculated as described in Section 4.7.2.

### 4.7.8 Boolean and Arithmetic Analysis

The Performance Tree formalism also provides operators for performing “arithmetic comparison”, “arithmetic operation”, “Boolean negation”, “Boolean disjunction”, “Boolean conjunction” and “In Interval” operations. To evaluate these nodes a suite of recursive procedures was implemented to evaluate subtrees with these node types. Each node is evaluated on a separate thread and results are passed up the evaluation tree as laid out in 4.1.

Although trivial to evaluate, these node types must still evaluated in the same generic fashion laid out in Section 4.1, the reason for this is that we cannot assume any prior knowledge of the structure of the performance tree being evaluated. Although an extremely large overhead is incurred, performing such analysis in the evaluation framework this is completely necessary to ensure any valid Performance Tree can be evaluated completely. Verification is performed to ensure the correct value nodes are input to operation nodes, taking into account that different operation nodes may return boolean or real valued results which are then used as input. For example the Percentile node returns a numerical value which is required to be handled in the same way that a numerical value node would be.

### 4.7.9 State Function specification verification

A State Function node represents a function applied to a state that returns a real number. A State Function \( \varepsilon \) is used as input to the SS:P node, this operator then calculates the steady-state probability of \( \varepsilon \) taking a particular value. The State Function expression which is specified by the user is used by the DNaMac (Section 4.7.1) tool where it is inserted into code, linked and compiled. For this reason the expression specified by the user must be verified to ensure it is of the correct form for two reasons:

1. As it is inserted into running code an invalid expression could cause a compilation error, the DNaMac tool would then not be able to run.

2. As the expression is entered directly into running code, users with malicious intent could formulate an expression to cause harm to the evaluation architecture and related systems.

In evaluating whether an expression is valid verification is performed to ensure that the expression conforms to the structure of a well defined grammar.

Statements in the grammar can be of two different forms:
4.7. Performance Tree Node Evaluation

- **Value Statements** use arithmetic operators and place labels to specify a function which will evaluate to a real number. Consider the following example which is used in Section 5.3 and applies the hospital model, Figure 5.1:

What is the steady-state probability distribution of nurses and doctors waiting for patients to arrive?

To obtain the numbers of nurses and doctors we use the following function:

\[
F(d, n) := \#(\text{doctors}) + \#(\text{nurses})
\]

- **Conditional Statements** behave like an if-then-else statement, with the resultant then and else clauses being Value Statements, the syntax for this statement mirrors that of the conditional assignment statement of C, C++ and Java\(^\text{TM}\). For example:

What is the steady-state probability distribution of nurses and doctors waiting for patients to arrive where we do not distinguish between values greater than zero?

The state function we require here is similar to the State Function above, except instead of collecting all the distinct values of nurses and doctors that we want to observe, we only distinguish between the cases where the number of nurses and doctors is zero and where it is not.

\[
F(d, n) := (\#(\text{doctors}) + \#(\text{nurses}) = 0) \ ? 0 : 1)
\]

The verification is carried out using \texttt{java.util.regex.Pattern} package which supports the specification of regular expressions and can then be used to check if a given string “matches” the regular expression. This validation is run on both client to ensure only valid statements are allowed to be submitted to the analysis tool. In this way users can formulate powerful expression which we can be sure will not cause problems in the evaluation architecture on the server-side.

**State Specification of SS:P Node**

The **SS:P** node has an optional value node specifying a set of states. This places a constraint on the set of states being considered. In evaluating such a specification we use the boolean condition of the state in a conditional statement (as described above) returning a sentinel value where the state condition is not satisfied and returning the State Function where it is satisfied. The distribution which will then be obtained from DNAmaca will have entries for all State Function values and also the sentinel value. The sentinel value will represent the probability of the specified state not occurring. As we are only interested in state configurations where our state constraint is satisfied we must discard the sentinel density and scale all other distribution values so the rest of the distribution sums to one. In this way we obtained the distribution of our state function constrained to a particular set of states.
Chapter 5

Case Study

To demonstrate our performance analysis pipeline, we will design queries and calculate relevant results for two example case studies based on a small modified version of the Accident and Emergency (A&E) department model introduced in [36].

5.1 Accident and Emergency Model

In the GSPN model of Figure 5.1 there is an initial group of healthy people who fall ill and go to a hospital – arriving either as walk-in patients or by ambulance. Walk-in patients await assessment by a nurse, while ambulance patients are given prioritised attention. Patients are subsequently either seen by a doctor for treatment, sent for lab tests, or taken for surgery. Once a patient is discharged from the hospital, (s)he is assumed to be healthy again. The model is parameterised with $P$, $N$ and $D$, which denote the number of tokens on the places healthy (people), nurses and doctors.
respectively. In the following examples, we set $P = 5$, $N = 2$ and $D = 2$, yielding an underlying Markov chain with 3,815 states and 16,530 transitions. In order to illustrate better the types of passage time query that can be asked, we have modified the model to include an extra place (“finished” on the diagram) which collects patients as they leave the hospital. Only when the entire population $P$ has arrived at this place will the immediate transition “reset” become enabled and repopulate the system with healthy individuals. This allows us to measure the time taken to process all patients in the system without the need to identify individuals (i.e. by tagging tokens).

5.2 Example 1

Example 1: Is it true that the 95th percentile of the time taken for all patients in the system to fall ill, complete treatment and be discharged from the hospital is less than 24 hours?

![Performance Tree query with PTD and Percentile nodes.](image)

The screen shots of the Performance Tree user interface for the query in Example 1 is given in Figure 5.2 and its evaluation window is given in Figure 5.3(a). Relevant state labels for this query are:

\[ \text{all patients healthy} := (\#(\text{healthy}) = P) \]

\[ \text{all patients treated} := (\#(\text{finished}) = P) \]
Figure 5.3: Case Study Example 1: screen-shots of Performance Tree Evaluation Module

As mentioned in the previous Section, the time range of interest for pdf and cdf plots is computed automatically. The result generated by the PTD operator for the hospital model in this case study is shown in Figure 5.3(b). The result generated by the Percentile operator is shown in Figure 5.3(c) and highlights the location of the percentile on the CDF plot. In the traffic light-based status indicator system shown in Figure 5.3(a), a green status indicates that results have been computed and are available for inspection (by clicking on the corresponding node).

It can be deduced from the results in Figure 5.3(c) that the 95th percentile of the time for all Accident and Emergency patients to be treated is 19.795 hours (rounded to 3 decimal places), hence the query evaluates to true.

The result in Figure 5.3(b) shows the probability of a patient leaving the A&E unit of a hospital at a certain time $t$, this is a graph of the passage time density. Figure 5.3(c) gives the probability of a patient leaving the A&E unit of a hospital within a certain time...
interval, this is a graph of the passage time distribution.

### 5.3 Example 2

**Example 2:** What is:

1. the average rate of occurrence of surgeries?
2. the steady-state probability distribution of the number of patients waiting for treatment?
3. Given that one or more patients are in surgery what is the steady-state probability distribution of nurses and doctors waiting for patients to arrive?

![Figure 5.4: Steady state and firing rate measure specification using Sequential operator for multiple queries.](image)

This example demonstrates three distinct features of Performance Trees: a steady-state probability calculation, a firing rate calculation and the ability to combine multiple independent queries into one. The Performance Tree for this query is given in Figure 5.4, which shows how independent sub-queries are combined using the ; operator. The first part of the query is seeking the average rate of occurrence of an action, which is represented by the **FR** node, the second part of the query addresses a steady-state probability distribution with the **SS:P** node, while the third demonstrates the full expressive power of the **SS:P** node specifying a specific state over which to apply the **State Function** node.

The relevant state function for query 2 is
5. Case Study 5.3. Example 2

\[ \text{no patients} := (\text{#(waiting to be treated)} + \text{#(waiting room)} + \text{#(trolley)}) \]

The relevant state function for query 3 is

\[ \text{no nurses and doctors} := (\text{#(doctors)} + \text{#(nurses)}) \]

The relevant state label for query 3 is:

\[ \text{patient in surgery} := (\text{#(Surgery)} \geq 1) \]

The results of this query are as follows.

Figure 5.5(a) shows the average rate with which surgery is performed is 0.181 (3 d.p.) operations per hour.

Figure 5.5(b) gives the number of patients awaiting treatment at steady-state. The average number of patients waiting to be treated at steady-state is 0.247 (to 3 decimal places), with a variance of 0.284 (also to 3 d.p.).

Figure 5.5(c) gives the number of nurses and doctors awaiting the arrival of a patient to treat given one or more patients are in surgery. The average number of nurses and doctors waiting is 0.426 (3 d.p.), with a variance of 0.918 (3 d.p.).
5.3. Example 2  

(a) **Firing Rate** results. Average rate with which surgery is performed.

(b) **Steady State Probability** results. Distribution of the number of patients waiting to be treated at steady-state.

(c) **Steady State Probability** results. Distribution of the number of nurses and doctors waiting for patients to arrive given one or more patients are in surgery.

Figure 5.5: Case Study Example 2: screen-shots of *Performance Tree Evaluation Module*
Chapter 6

Evaluation

In this Chapter we evaluate our evaluation architecture for Performance Trees, this is done in three different ways:

- Existing tools from Section 2.6 are compared with our architecture. This comparison focuses on functionality offered by each of the tools, including support for different types of analysis and the way in which results are provided to users.
- We discuss the degree of support for the full suite of Performance Trees.
- To give an impression of performance we perform three benchmarking tests; these evaluate the Performance Tree evaluation architecture with multiple clients, varying processor allocation and in comparison to the old architecture.

6.1 Comparison with Existing Evaluation Tools

In Section 2.6, some existing performance analysis tools were introduced. The applications of these tools are comparable those of PIPE and the evaluation architecture, and so a comparison of their functionality with PIPE’s will help us to highlight any strengths or weakness of the current implementation. Observations are as follows:

- The evaluation architecture realised in this project is similar in function to that of the PRISM tool; model and query are composed on a client based tool, written in Java\textsuperscript{TM}, and sent for analysis on a remote cluster with bespoke Analysis Tool, written in C++, performing calculations. This provides support for the way in which our architecture has been implemented.
- Both PRISM and Möbius support multiple modelling and querying formalisms. This implies both system flexibility and support for the preferences of users with respect to the specific formalism they may choose. PIPE lacks some of this flexibility and currently supports only Petri net models and Performance Tree queries. As one of the aims of this project was to facilitate simple composition of models and queries this is a shortcoming in our implementation (however this is soon to be addressed 7.3.5).
6.2 Assessment of Functionality

By making use of Performance Trees and the automatic natural language query interpretation system PIPE is a more accessible tool for non-expert users composing performance queries; this was one of the intentions of the Performance Tree formalism. Both Möbius and PRISM use text based query specification this requires users to possess a more in-depth knowledge of the formalism they choose to use such as CSL. Non expert users will find it easier to specify measures using the PIPE tool.

6.2 Assessment of Functionality

This project aimed to provide complete support for the suite of Performance Tree nodes outlined in Table 2.1. Due to the relative infancy of the Performance Tree formalism, the suite of operators evolved with the progression of this project. As discussed, a new operator, Percentile, was introduced. This took focus away from the implementation of other operators which became more peripheral to the formalism in general. The operators that are not currently supported for evaluation using PIPE are Convolution, Probability in States and States at Time. The convolution and states at time operators are esoteric to the Performance Tree formalism, the types of measures they allow one to express are not widely used. Probability in States is a transient operator; transient state probabilities are 1 of the 8 patterns highlighted in [24] and formulate 6 of the 154 example queries presented. A lack of support for this type of query demonstrates a shortcoming in the current implementation (a discussion of how this could be rectified is provided in Section 7.3).

To put these observations into perspective we observe that the current implementation supports the evaluation of the 4 most prevalent patterns, probabilistic invariance; probabilistic existence; probabilistic until; probabilistic precedence, these account for \(\approx 93\%\) of 154 example queries in the sample. This demonstrates the vast majority of queries that a performance engineer may wish to specify are expressible using PIPE and can be evaluated on our architecture. Over the course of this project we introduced the Percentile node to the formalism, this node provides users with a more convenient way to express the probabilistic existence and probabilistic until patterns, these account for \(\approx 56\%\) of the queries in the survey, this goes further to demonstrate not only the expressive power of this formalism but also it’s ease of use.

6.3 Benchmarking

To facilitate the benchmarking outlined below the logging system detailed in Section 4.4 was used to provide response times accurate to one thousandth of second. This provided an unintrusive mechanism by which we could measure the time elapsed between the analysis server first acknowledging an evaluation request and the evaluation completing with results having been sent to the client.

6.3.1 System performance with multiple clients

The analysis architecture is designed to have multiple concurrent clients at any one time. The system was tested with 1 to 5 concurrent clients evaluating the query in
Section 5.2 on the accident and emergency model of Figure 5.1. During this test we disabled the model-query cache to ensure results were not shared between tasks. Results are taken as an average of 5 individual evaluation sessions.

![Graph showing time taken to evaluate a query with multiple clients](image)

Figure 6.1: Graph showing time taken to evaluate a query with multiple clients

The general pattern observed from Figure 6.1 is that as the number of concurrent clients increases, evaluation time increases in linear fashion. It can be seen that when there are only 2 concurrent clients executing that the full performance degradation is not felt as evaluation time rises from 40 to only 60 seconds. This can be accounted for by the fact that only 24 processors are being used per client, hence, with 2 clients the full 48 processors available on the Camelot cluster are being used and an allocation occurs which supports independent processor allocation between the tasks. There is still some performance deterioration experienced as the communication channels have twice the traffic as with one client.

### 6.3.2 System performance with varying processor allocation

The analysis architecture allows a user to specify the number of processors to distribute their analysis task across when using the Analysis Tools. The system was tested with 6 to 48 processors in increments of 6 processors evaluating the query in Section 5.2 on the accident and emergency model of Figure 5.1. Results are taken as an average of 5 individual evaluation sessions using 24 processors on the camelot cluster.
As can be seen from Figure 6.2, evaluation times decrease from around a minute with 6 processors, to around 40 seconds with 30 processors, above 30 processors a significant penalty is incurred in the evaluation time. One may expect evaluation time to decrease as more processors are used, however, the results do not reflect this. The reason is that between 30 and 36 processors the overhead incurred in distributing the job begins to outweigh the time gained by dividing the task across an extra 6 processors; as the number of processors reaches 48 this overhead is exacerbated to the point where it takes twice as long to perform the evaluation with 48 processors as it does with 6. It should be noted that the optimal processor allocation is highly task specific and related to the size of the model. It is for this reason that the number of processors to be used is a client specified variable.

### 6.3.3 Comparison with old architecture

To get an impression of the overhead incurred by using the new architecture a comparison of the new and old architecture was made. As both architectures make use of the same Analysis Tools this benchmark would provide a good impression of the performance of the new components of the architecture including the Performance Tree Evaluation Module, Result Display Module, Performance Tree Division and Dependency Analysis and Analysis Thread Initialisation. Two types of query were used to perform
the comparison, a Steady State query and Passage Time query. Although the queries are specified differently using the Performance Tree Editor to the Steady State and Passage Time analysers, the semantics are identical.

**Steady State Analysis**

The query used was part 1 of the query introduced in Section 5.3 on the accident and emergency model of Figure 5.1. Results are taken as an average of 5 individual evaluation sessions. The mean evaluation time for the new architecture was 11.738 seconds with a variance of 3.053. The mean evaluation time for the old architecture was 10.061 seconds with a variance of 2.746.

**Passage Time Analysis**

The query used was the passage time element of the query introduced in Section 5.2 on the accident and emergency model of Figure 5.1. Results are taken as an average of 5 individual evaluation sessions. The mean evaluation time for the new architecture was 36.616 seconds with a variance of 30.174. The mean evaluation time for the old architecture was 35.677 seconds with a variance of 24.235.

**Consideration**

As can be seen there is little difference between the architectures in the time taken to evaluate both types of queries, however, this could be expected as the same analysis tools are being used on the camelot cluster to perform the evaluation. This benchmark provides insight into the overheads of the new architecture which can be observed to be around 1-2 seconds. This increased delay can be attributed to the increased data transmission in the new architecture from the server to the client in the form of interim results and notifications on the progress of the query. This would also account for the lack on an increase in the delay for the Passage Time Analysis, which takes longer to execute than the steady state analysis implying the delay is not related to the total evaluation time of the query. Due to the fact that the new architecture supports the specification of more expressive queries that can be expressed in a more accessible manner we argue that this small overhead is entirely acceptable.

**6.4 Summary**

In this Chapter, we compared our implementation with other tools offering similar functionality, highlighting the key advantages and limitations of our evaluation architecture with respect to the current state-of-the-art. A discussion of the support for Performance Tree operators was provided, 93% of queries which users may want to ask of their system are now supported by the architecture, work has begun to support the rest as demonstrated by the development of the CONE tool and the integration of HYDRA into the architecture. We performed benchmarking to get an impression of how our architecture performed with differing client loads and varying processor allocation. Finally, to gain insight into how the new architecture performs in comparison to the old one we performed some like for like testing, this demonstrated a small and overhead is
incurred through use of the new architecture; however, due to the richer variety of queries that we can now compose this overhead is entirely tolerable.
Chapter 7

Conclusion

7.1 Summary of Achievements

In this paper, we have introduced a distributed evaluation environment for performance queries expressed as Performance Trees on GSPN models. We have presented the query design and analysis workflow, and have discussed in detail features of the client-facing front-end, as well as the client-shielded analysis back-end. We have described how the popular PIPE tool has been extended to support the graphical creation of Performance Tree queries, and how these queries are evaluated on a parallel and distributed analysis cluster. We have described the components of this integrated analysis environment, and have elaborated in detail on the evaluation aspects of performance queries. We have also demonstrated – in the context of a case study of a hospital's Accident & Emergency unit – the complete process of designing a GSPN model of the case study environment and an applicable Performance Tree query, submitting these to the analysis back-end and eventually obtaining and inspecting relevant results. The case study example illustrates how our toolset can be used to obtain results relating to patient arrival-to-discharge time and utilisation of hospital employees.

The work conducted during the course of this project contributed to the publication of 3 research papers [16–18] details of which can be found in Section 1.3. The PIPE: Petri net editor [1] is an open source tool, the client-side modules implemented as part of the evaluation architecture will be included in the next release of this tool; this will allow users to design models, compose queries and evaluate them on the Camelot cluster from around the world.

7.2 Discussion of Applications

The evaluation architecture implemented in this project has significant practical applications. In particular, one could apply this functionality in the following areas:

- In many branches of engineering response times are of crucial importance. For example, Telecommunications service providers are required to provide response time guarantees for their wireless services; the 3G standard requires a guaranteed response within a given time-frame to operate correctly. Therefore, telecom
companies require large-scale QoS modelling and analysis.

- Various service based industries utilise response time analysis to gain insight into fulfillment of customer requirements. Public health care providers are one such example, they require response time analysis of patient flow models to help improve patient-perceived QoS; this is of ever increasing importance in the UK where measures such as these are used, in-part, to determine hospital funding from the NHS. See Chapter 5.

### 7.3 Further work

This sections lays out potential extensions to the architecture that has been implemented that would formulate a valuable addition to the current tool set. Many of these extensions are inspired by past and current research performed at Imperial College in the Department of Computing.

#### 7.3.1 Support for all Performance Tree Operation Nodes

Currently there are 3 operation nodes which are not supported by the evaluation architecture.

- Convolution
- Probability in States
- States at Time

With a substantial architecture now in place providing complete support for unsupported operators would be a straightforward task. In fact, the CONE and HYDRA tools, supporting the Convolution and Probability in States nodes respectively, are fully implemented as stand alone command line tools, this is why they are included in the architecture diagram in Figure 3.2. Work is required to fully migrate these tools into the architecture.

#### 7.3.2 Further optimisations for Performance Tree evaluation

There is some scope for increased optimisation of the scheduling of computations in order to achieve improved analysis cluster response times. One such optimisation would be to recognise identical elements in a Performance Tree only need to be evaluated once. This optimisation is best illustrated with the use of an example query one may wish to ask of a model.

- **What is the coefficient of variation\(^1\) of the time for a patient to be seen, treated and discharged from the hospital?**

As can be seen from Figure 7.1 the query requires 3 instances of an identical passage time density to be calculated, this is only required to be calculated once. A simple way to

---

\(^1\)the ratio of the standard deviation to the mean
7. Conclusion

7.3. Further work

recognise redundant calculations such as this would be to generate a hash of the file containing the analysis tool specific representation of the query using a hashing algorithm with a very low probability of clashes. When matches between hashes were found only one analysis task would be sent to the cluster. Figure 7.1 also demonstrates the usage of a Macro node to abstract away complex concepts from the query representation.

7.3.3 Natural Language Query Specification

Natural Language is already used in the Performance Query Editor to aid query composition. This usage could be extended with the introduction of a natural language query specification mechanism. This would be used to build a query using natural language stubs, guiding the user through the query specification process, as stubs were selected the tree would be built automatically. This would allow users with no expertise in the area of performance analysis to compose powerful queries in a simple and intuitive manner. This idea is in part based on work in [24] where a “Structured Grammar” is theorized which could be used to build performance queries directly. However, it is suggested that CSL is the formalism through which queries are specified and not Performance Trees as would be the case in PIPE.

7.3.4 Tagged Customer Support

Tagged Customers [36] are used to construct performance queries relating to individual tokens in a Petri net model. In the Petri net context this means tracking the flow of a token through throughout the model. This necessitates the introduction of an extra transition for each existing transition in the net to differentiate between the cases of forwarding tagged and untagged customers. To specify queries involving a tagged customer, the \textit{token@place} notation is used, which is an atomic proposition that is
attributed to a state if the token representing the tagged patient is at place place in the model.

Incorporating this concept into the evaluation architecture would provide a mechanism allowing us to construct more powerful performance queries, for example relating to tracking the progress of individual patients in the Accident and Emergency model.

### 7.3.5 SPA model extension

We are currently in the process of integrating support for PEPA-based stochastic process algebra model specification into our analysis toolset. This functionality would allow users a choice in the modelling formalism used to specify their models. Support for the viability of this work could be taken from the observations in Section 2.6, where it is observed that other prominent Performance Analysis tools support many modelling formalisms, allowing the performance engineer to use the formalism best suited to their task. As PEPA models can be mapped onto CTMCs, in similar fashion to GSPN models (see Section 2.1.4), an SPA model extension in PIPE could make use of the analysis tools currently in place so long as a mechanism for the translation of SPA models to DNA macro input language was implemented.
Appendix A

Extra Case Study Screen shots

These are extra screen shots from the case study in Chapter 5, Section 5.3.

Figure A.1(a) shows the textual results of the steady state distribution of patients waiting for treatment. Figure A.1(b) shows the steady state distribution nurses and doctors given a patient is in surgery. Mean, variance and standard deviation are provided, this is the data from which Figures 5.5(b) and 5.5(c) are generated. This is also the data which would be used to calculate the Moments of the distribution, see Section 4.7.4.

(a) Steady State Probability Distribution of the number of patients waiting to be treated at steady state.
(b) Steady State Probability Distribution of the number of nurses and doctors given at least one patient is in surgery.

Figure A.1: Case Study Example 2 screen-shots of Performance Tree Evaluation Module
Bibliography


