Imperial College London
Department of Computing

A unified performance query formalism

Matej Kohut
MEng Individual project - Final Report

Supervisor: Dr. Jeremy Bradley
Co-Supervisor: Dr. Giuliano Casale
Submitted in June 2012

Abstract

Stochastic Process Algebras can be used to model complex systems. Due to their direct mapping to continuous-time Markov Chains, it is possible to analyse them and obtain steady-state probabilities for finite models. Recent research has also come up with the Unified Stochastic Probes formalism [24], which also allows monitoring the behaviour of such systems quickly and reliably, while maintaining simplicity of expression.

In this project we explore the ideas for making this formalism widely accessible and better applicable for users. We attempt to describe a mapping from Unified Stochastic Probes to a graphical representation compatible with Performance Trees [30] and discuss alternative directions. We compare this to the original way of specifying passage time queries in Performance Trees formalism and informally show that Unified Stochastic Probes surpass this definition. Furthermore, the syntax is more intuitive and convenient.

We provide a full implementation of Unified Stochastic Probes formalism parser, along with the support of passage time calculations using both simulation and fast fluid flow approximation techniques. We also demonstrate the usefulness of the formalism on a range of examples and compare our results to manually hand-crafted results from previous work.

As a side project, we introduce extensions to GPAnalyser open-source software, which increase its modelling capabilities. We extended it with passive, weighted passive and immediate actions. Additionally, by substituting Java dynamic compilation for C++ native libraries dynamically loaded with JNI, we significantly increased the tractable state-space.

Acknowledgements

I would like to express my extreme gratitude to all the people who made this project possible. Namely:

- Dr. Jeremy Bradley, my supervisor, for the project idea and all the advices, insights and enthusiastic support throughout the whole course of the project.
- Dr. Giuliano Casale, my co-supervisor, for all the hints and ideas for the report and the overall support.
- Dr. Richard Hayden for the very detailed information about this fascinating topic.
- Anton Stefanek for all the assistance with the GPAnalyser software and its extending.
- All the personnel of Imperial College London, who were teaching me, for everything they gave me.
- My family and friends, who have supported me throughout the studies and helped me always when I needed it most.

Contents

List of Figures

Chapter 1

Introduction

When dealing with any system, we are usually interested in its performance and reliability. Often it is part of an SLA (service level agreement) for clients, who require certain performance standards to be met. It is thus increasingly important to evaluate performance of systems easily and reliably, and to provide these assurances. The current performance evaluation techniques of system models are more and more powerful. They allow users to find various performance and reliability metrics (queries) of modelled systems; e.g. in your particular system, how fast will some job be finished or how probable it is in long-term that a particular server fails or even how many servers will serve at least 50 clients in the next 10 minutes. However, it is quite hard to specify any more advanced queries. These require users to write a lot of code in the modelling language (modelling formalism) of the original model, even though users have no real need to know all the low-level details (i.e. they do not need to know the modelling language). This project aims to simplify the process of defining performance queries by providing an implementation of a convenient formalism called *Unified Stochastic Probes* (Section 2.5). In addition, these are further developed and integrated into a graphical performance queries representation called *Performance Trees* (Section 2.8).

1.1 Motivation

The state of the art of performance evaluation allows complex and detailed performance metrics to be taken when dealing with abstract system models. Fluid approximation techniques already simplify the simulation of such systems to mere mathematical evaluation of sets of ordinary differential equations, which takes incomparably shorter time.

One of the most important and interesting performance metrics are passage time queries. These express a probability of exhibiting certain system behaviour at various times. For example, how long will it take till 10 different clients download some data from the servers. Another example is the time until one particular client downloads data 10 times if the system is already in the steady-state (equilibrium). Unfortunately, powerful techniques for performance estimation currently available are not sufficient for wider user adoption. The main problem is that the users need to know a lot about a particular modelling formalism (e.g. PEPA or other stochastic process algebras, Stochastic Petri Nets, Stochastic π-calculus, etc.), in which the model of the system is defined. They also need to know how to translate their performance queries to the same formalism and compose them with the system model. They do not require this knowledge to understand the system or the metrics they are interested in though. In addition, augmenting the models directly to obtain the performance metrics is a discouraged practice, since it obscures the model. In theory, the performance counting components could be written as passive observers of the model, which is essentially what most of the existing performance query formalisms do. Hand-crafting the queries is thus not only very complicated and error-prone, but also pointless.

The previous research has come up with the Unified Stochastic Probes formalism which aims to offer a convenient way of specifying passage time queries. These are independent of formalism and much simpler to define, use and comprehend. However, the experience shows that for users it is most intuitive when they are allowed to use their visualisation. For this, the Performance Trees notation, which allows users to graphically express both quantitative and qualitative queries, is the perfect fit. It is also a highly abstract notation which grants users the ability to define complex queries with much less effort. We therefore decided that Performance Trees should be provided as a user-friendly solution to this problem.

To simplify the development, we have decided to implement the Unified Stochastic Probes formalism as an extension of an open-source project for analysing massively parallel systems, GPAnalyser ([29]). This project utilizes its powerful backend system for fluid flow analysis or simulation, which we built upon.

The novelty of this project is in completely implementing the Unified Stochastic Probes formalism, both syntax and evaluation, and describing how can they be linked to the Performance Trees formalism to offer a simple graphical and very powerful formalism for passage time queries.

1.2 Objectives

This project had three main objectives.

- To introduce a complete working Unified Stochastic Probes parser implementation as described by Hayden [22].
- To build a passage time computation module for GPAnalyser using the Unified Stochastic Probes, also described by Hayden.
- To introduce the graphical syntax of Unified Stochastic Probes for using with Performance Trees [30] directly.

1.3 Report structure

This report is further structured as follows:

- Chapter 2 gives an introduction to the background theory.
- Chapter 3 describes the translation of Unified Stochastic Probes to the Performance Trees formalism.
- Chapter 4 explains how the project was implemented as an extension of GPAnalyser.

- Chapter 5 shows the capabilities of the project on real-life models. It also concludes this work, compares with the previously hand-crafted results and discusses the overall achievements of the project as well as the future work.

1.4 Contributions

This project has multiple contributions divided into two categories.

- Theoretical:
  - Proposes a way of linking the Unified Stochastic Probes to the Performance Trees formalism (Chapter 3).
  - Shows that this is superior to the originally proposed Performance Trees passage time calculations (Appendix A).
- Implementation (Chapter 4):
  - GPEPA parser in GPAnalyser was extended with passive, weighted passive and immediate actions, thus making it a full iGPEPA Parser. Furthermore, the algorithm for *vanishing states removal* was introduced.
  - A complete parser for Unified Stochastic Probes formalism and algorithms for their translation to iGPEPA.
  - Passage time computations in both simulation and fluid flow approximation mode using the Unified Stochastic Probes.
  - The capabilities of GPAnalyser were extended to very large models. This is further explained in the Section 4.8.

1.5 Publication

During the course of this project, a tool paper was written and accepted for the QEST 2012 conference. This is a short tool paper describing the implementation contributions of this project. The paper is attached in Appendix C and will be presented at the QEST conference.

Chapter 2

Background

2.1 Introduction

The main objective of this project was to simplify the computation of passage times in system models. We believe the best simplification for the user is if they can visualise the issue. The best way to provide such a help is to use graph representation of the performance query. In our case, we chose Performance Trees formalism.

There were many approaches possible. We considered translating performance queries from graphical (Performance Trees, the Section 2.8) representation directly to the process algebra, i.e. in our case iGPEPA (Section 2.4). However, direct translation would diminish the Performance Trees advantage of abstraction from any particular process algebra. Furthermore, Performance Trees are a very high level formalism and a direct translation to the desired process algebra might require significant effort and development time. Even though that could be a great contribution in itself, the same effort would have to be made for every other process algebra we would like to use.

The best way how to approach this issue is to make a layered system. Performance Trees can build upon a less high-level formalism x. Formalism x could in theory be built on top of another less powerful formalism y, etc. The last layer would be the concrete process algebra. In our case, we decided one middle level should be sufficient, for which we chose the Unified Stochastic Probes formalism (or just Probes for simplicity), which we describe in the Section 2.5.

Unified Stochastic Probes are in theory also an abstract concept independent of the process algebra. There has been a lot of research invested into the translation of Probes to a process algebra called iGPEPA, which is a derivation of process algebra PEPA (the Section 2.3). Since there are many tools, which work with PEPA, it was decided it will be best to continue in this work and use iGPEPA in our case as well. The translation of Probes is shown in the Section 2.6.

2.2 Related work

There have been other formalisms proposed for specifying the performance queries easily. The most popular ones are CSL [12] and its derivatives, which we introduce below. Apart from that, there are other implementations of Performance Trees and we talk about one particular.

2.2.1 CSL and derivatives

Continuous Stochastic Logic (CSL) is the most widely used logic for specifying performance queries on Continuous Time Markov Chains (CTMCs) directly. It is popular, because its basic rules are very simple and many tools already support it. However, as described in [30, Section 5], it can be easily subsumed by the more powerful Performance Trees formalism, which offers much more abstract notions. Also, it is just another textual formalism, which has no visual aid for the user.

There are other formalisms, which try to extend CSL and increase its power. Notable are *aCSL*, *asCSL* and *eCSL*.

Formalism aCSL [26] extends CSL with expressing simple requirements on taken actions. The basic premise is, that it is more natural to measure the overall behaviour of the system, rather than just various state properties at certain times. This formalism is therefore much closer to the idea of Performance Trees and queries possible with it can express much more than CSL.

Another formalism subsuming both CSL and aCSL is asCSL [14]. It provides new options for specifying action sequences. It is not as advanced as any of the probes mechanisms introduced in Section 2.5, but with this formalism, behaviour-oriented performance queries are already possible.

Last formalism we could use is eCSL ([15]). This formalism is working with higher-level models called Semi-Markov Stochastic Petri Nets (SM-SPN). It is powerful and allows transient and passage time queries, including constraints. However, its strong reliance on SM-SPN is a disadvantage and we prefer Performance Trees for being an abstraction over modelling formalisms.

2.2.2 Performance Trees implementation in the PIPE tool

An open-source PIPE tool [9] was extended to support the full-range of Performance Trees operators. As described in [16], it is an impressive work providing the performance evaluation queries with a nice GUI toolkit. PIPE works with modeling formalism called *Generalised Stochastic Petri Nets* (GSPN), which is the most important difference from our project. We should note, that GSPNs are, after removing all immediate transitions, convertible to a CTMC. Since we can achieve the same with iGPEPA, which also supports immediate transitions, the expressible power of both formalisms is roughly the same. Petri Nets are considerably harder to design without a GUI, whereas iGPEPA is easy to master in a textual representation. Another difference from PIPE is in dealing with complexity of simulations and their performance. PIPE supports simulation of Performance Trees queries on a high-performance cluster. We mainly use fluid-flow approximation instead of simulation, which is an approximate, but computationally faster technique and provide simulation only for comparison.

2.3 PEPA process algebra introduction

PEPA stands for Performance Evaluation Process Algebra and is a stochastic process algebra, which is designed for modelling systems as complex state machines. Its syntax is very simple and thus it is quite popular and there are many tools, which support it. Stochastic means it is not deterministic when choosing the next state in the process. The next state is chosen based on the negative exponential probability distribution, which, in form of rate, determines the next performed action of a component. Actions essentially represent transitions - moving from one state of the component to another, although it is possible, that after executing an action, a component stays in the same state. The action to be executed is chosen by racing the currently available actions in the component. Informally, with higher rate, action is likely to occur sooner.

Another advantage of PEPA models is, that they can be straightforwardly translated to continuous-time Markov chains (CTMC) [13]. There are known efficient numerical solutions for steady-state probabilities of CTMCs, although they are computationally feasible only for small chains. Unfortunately, PEPA models with lots of replicated components are usually mapped to very large CTMCs.

There have been a lot of materials about PEPA. Formal specification of PEPA operational semantics can be found in [28, Chapter 20]. For our purposes, we do not need so detailed information. All the information required will be summarised below. This section was inspired by a similar introduction from [24, Section II.A].

The basic PEPA grammar can be expressed with these two rules:

$$\begin{aligned} S &::= (\alpha, r).S \;\mid\; S + S \;\mid\; \boldsymbol{C}_S \\ P &::= P \underset{L}{\bowtie} P \;\mid\; S \;\mid\; \boldsymbol{C}_P \end{aligned} \tag{2.1}$$

where $\alpha \in \mathcal{A}_t$ is a *timed action* (or transition) and $L \subseteq \mathcal{A}$, where $\mathcal{A}$ is the set consisting of all action types (in case of PEPA only $\mathcal{A}_t$). Parameter $r \in \mathbb{R} \cup \{n\top \mid n \in \mathbb{Q}, n > 0\}$ is a rate parameter.

There is intentionally a clear distinction between P and S. Component (or process) S is intended to be a simple *sequential component*, whereas P is a *parallel component*. The syntactic separation allows cooperation between sequential components only. Symbols $\boldsymbol{C}_S$ and $\boldsymbol{C}_P$ represent constants for sequential or parallel components respectively.

We took the liberty to ignore some more advanced operations (e.g. action hiding), which can be added later. The basic rules therefore are:

- *Prefix* specifies a possible transition to another component state. It is in the form $(\alpha, r).S$, which says that after executing the action α, the process will advance to the state S. We say the process currently *enables* the action α. As we have mentioned, r is the rate for the exponential distribution, which determines the duration of α action.

- *Constant* is a label for a component state. The notation is $\boldsymbol{X} \overset{def}{=} P$. It allows a recursive definition, so $\boldsymbol{X} \overset{def}{=} (\alpha, r).\boldsymbol{X}$ is a valid component, which will execute action α with rate r forever.

- *Choice* $P + S$ means that the component can behave either as P, or as S. It basically races these components and behaves as the one, which is prepared to execute its action earlier. Therefore; $(\alpha, r_\alpha).S + (\beta, r_\beta).Q$, will execute either α or β, depending on the rates and also cooperation with other components, as we will explain below.

- *Cooperation* is a way of expressing that two components work in parallel. It may be the case, that some of the actions must be synchronized between them. The syntax is $P \underset{L}{\bowtie} Q$, which says that components P and Q are executed in parallel and must synchronise on actions in the set L. Actions enabled by these components and not in the set L are raced in the usual way disregarding the other cooperating component. Set L may be empty, in which case these components work completely independently. We denote that as $P \parallel Q$, which is an abbreviation for $P \underset{\emptyset}{\bowtie} Q$. Another useful abbreviation is independent cooperation of n components of the same type (e.g. for component P) $P[n]$.

We define the *apparent rate*, $r_\alpha(P)$ of an action as an overall observed rate with which a component executes a timed action. Apparent rate is the only visible rate to the cooperating component and determines the rate of the cooperation. It is defined as:

$$\begin{aligned} r_\alpha((\beta,\lambda).P) &:= \begin{cases} \lambda & \text{if } \beta = \alpha \\ 0 & \text{if } \beta \neq \alpha \end{cases} \\ r_\alpha(P+Q) &:= r_\alpha(P) + r_\alpha(Q) \\ r_\alpha(P \underset{L}{\bowtie} Q) &:= \begin{cases} \min(r_\alpha(P),\, r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{if } \alpha \notin L \end{cases} \end{aligned} \tag{2.2}$$

Once an activity in the set L, or *shared activity*, is enabled in one component in Cooperation, it cannot be executed until both of the cooperating components enable this activity. When both of them enable it, enabled actions in each component are raced in the usual manner with a rate determined by the slower component - we assume *bounded capacity*, which means, that no component can execute an action faster than its own rate for that action. Basically, if a component is executing an action in synchronisation with another component, the whole activity is finished only when the slower component finishes.

There are also components with *passive* activity. That means the component enables the action specified, but can perform it only in cooperation with another component. This component therefore needs no rate for an action and we write the passive rate as $\top$ (e.g. $(\alpha, \top).P$). The rate is then determined during the cooperation as the apparent rate of this action from the other component. It is important to note, that cooperation between two components on an action, which is passively enabled in both of them, is forbidden. A standard PEPA component is also prohibited to enable the same action both actively and passively.

For a PEPA component P, we can define its set of *derivative states* ($ds(P)$), which is the set of all states of this component, reachable by taking any sequence of timed actions.

We also have to mention, that a Choice component can have multiple Prefixes with the same passive action α. These are then attached some weight. When this component takes α in synchronization with another component, it is probabilistically chosen which Prefix will be used depending on the weights. The formal meaning of weights is defined below.

$$\begin{aligned} m\top < n\top &\quad : \quad \text{for } m < n \text{ and } m, n \in \mathbb{Q} \\ r < n\top &\quad : \quad \text{for all } r \in \mathbb{R}, n \in \mathbb{Q} \\ m\top + n\top = (m+n)\top &\quad : \quad m, n \in \mathbb{Q} \\ \frac{m\top}{n\top} = \frac{m}{n} &\quad : \quad m, n \in \mathbb{Q} \end{aligned}$$

where $n\top$ is the abbreviation for $n \times \top$.

2.4 iGPEPA

iGPEPA is actually introducing two different extensions to the PEPA (the Section 2.3): *iPEPA*, which can use immediate actions and *GPEPA*, which can use grouped components. We will introduce them both in the subsections below. This section was inspired by a similar introduction from [24, Section II.A].

2.4.1 iPEPA

iPEPA is a simple extension of original PEPA operational semantics. It introduces a new type of prefix - *immediate prefix* with an *immediate action* a. If a component (process) enables such an action, it is executed instantaneously. With this new prefix, the augmented basic syntactic rules of iPEPA are:

$$\begin{aligned} S &::= (\alpha, r).S \;\mid\; [\mathsf{a}, w].S \;\mid\; S + S \;\mid\; \boldsymbol{C_S} \\ P &::= P \underset{L}{\bowtie} P \;\mid\; S \;\mid\; \boldsymbol{C_P} \end{aligned} \tag{2.3}$$

Formal operational semantics can be found in [24, Section II.A].

The syntax of immediate prefix with an immediate action is $[\mathsf{a}, w].S$, An enabled immediate action always takes precedence over any enabled timed actions. The parameter w is the weight of the immediate action and is used when there are multiple enabled immediate actions in the current component, in which case the next performed immediate action is chosen probabilistically using all enabled immediate actions weights. A useful shorthand is $\mathsf{a}.S$, which is the same as $[\mathsf{a}, 1].S$. Since cooperation between immediate and timed actions is not a well-defined operation, we prohibit it in iPEPA. Therefore, to make the distinction explicit, timed actions are drawn from set $\mathcal{A}_t$, immediate actions from set $\mathcal{A}_i$. Therefore, in iPEPA we have $\mathcal{A} := \mathcal{A}_t \cup \mathcal{A}_i$.

Also, in iPEPA, the derivative states of a component P, $ds(P)$ is a set of all states reachable by taking any immediate or timed actions.

An iPEPA component can still be translated to a CTMC. However, we require it to be *well-behaved*. A well-behaved iPEPA component satisfies these two conditions:

- *freedom from immediate cycles*, meaning that there is no cycle of immediate actions in the component. Simply put, in one component, we cannot travel from any state back to itself using only immediate actions.

- *deterministic initial behaviour*, meaning that there cannot be more than one path of immediate actions from the initial component state.

For translation of iPEPA component $\boldsymbol{P}$ to an CTMC, we can use its set of *non-vanishing* derivate states $ds^*(P)$. A non-vanishing state is a state which enables no immediate actions. In order to eliminate immediate actions from an iPEPA component, we can do *vanishing state removal.* This operation is in detail described in [24, Appendix B]. Basically, we do three simple steps.

1. If a state $\boldsymbol{S}_t$ can reach a state $\boldsymbol{S}_i$, emanating an immediate action a, by taking a timed action α, α will get a as a complementary action executed right after α. The next state $\boldsymbol{S}_x$ will be the one where the a would originally lead from the state $\boldsymbol{S}_i$. If $\boldsymbol{S}_x$ has any immediate action b, the process is repeated, so after executing α and a, b is performed leading to a state $\boldsymbol{S}_y$ as b would lead from $\boldsymbol{S}_x$, etc.
2. Remove the states with immediate actions.
3. New rates of the timed actions with complementary immediate actions are their original rates multiplied by the multiplied weights of all their complementary actions. Formally, $R := r \times \prod_{i=1}^{K} z_i$, where z_i are the weights.

This is illustrated in the Figure 2.1.

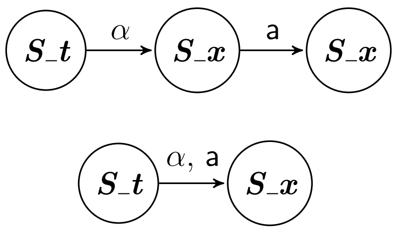

Figure 2.1: Vanishing state removal simplified process

F-components

An f-component, in some older works called a fluid component, is any standard iPEPA component. For fluid flow approximation, we require all f-components to be well-behaved.

2.4.2 GPEPA

GPEPA or Grouped PEPA is a simple extension of PEPA. It allows to have *component groups* with labels. A component group is a named cooperation of some components, or it can be a single component. Formally, the grammar of a labelled group D is:

$$D ::= D \wr\wr D \;\mid\; P \tag{2.4}$$

where $\wr\wr$ is unsynchronized cooperation between f-components and P is an f-component.

A *grouped PEPA model* is a model formed by cooperation between component groups. For every component from one group, it has to synchronize with a component from the other group on the actions specified in the set L. Formal syntax follows.

$$G ::= G \underset{L}{\bowtie} G \mid Y\{D\} \tag{2.5}$$

where Y is a group label unique for each group.

The $\wr\wr$ is essentially equivalent to $\|$ operation. However, we use two distinct combinators to resolve any possible ambiguities between parallelism of the groups and f-components inside these groups, which can have their own internal parallelism. The cooperation between the component groups is forbidden to synchronize on passive transitions. That means, every action, on which these two groups synchronize, must be internally associated with a rate (either it is a timed action, or passive which is always internally synchronized with some timed action).

Finally, in the table Table 2.1, we introduce some useful functions for working with GPEPA.

$\mathcal{B}(G, H)$	The union of all non-vanishing derivative states of all f-components in the component group of G which has group label H, e.g. $\mathcal{B}(\boldsymbol{CP}(N_c, N_p), \textbf{Consumers}) = \{\boldsymbol{C}, \boldsymbol{C\_get}, \boldsymbol{C\_use}\}$.
$\mathcal{B}(G)$	The set of all pairs whose first element is a component group label, say, H and whose second is an element of $\mathcal{B}(G, H)$, e.g. $\mathcal{B}(\boldsymbol{CP}(N_c, N_p)) = \{(\textbf{Consumers}, \boldsymbol{C}), (\textbf{Consumers}, \boldsymbol{C\_get}), (\textbf{Consumers}, \boldsymbol{C\_use})$, $(\textbf{Producers}, \boldsymbol{P} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T}), (\textbf{Producers}, \boldsymbol{P\_ready} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T})$, $(\textbf{Producers}, \boldsymbol{P\_done} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T})$, $(\textbf{Producers}, \boldsymbol{P} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T\_get})$, $(\textbf{Producers}, \boldsymbol{P\_ready} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T\_get})$, $(\textbf{Producers}, \boldsymbol{P\_done} \underset{\{get\_product\}}{\bowtie} \boldsymbol{T\_get})$.
$\mathcal{S}(G, H)$	The size of the component group with label H. That is, the number of f-components in the group, e.g. $\mathcal{S}(\boldsymbol{CP}(N_c, N_p), \textbf{Consumers}) = N_c$.

Table 2.1: Frequently used iGPEPA notation, where $\boldsymbol{C}$ is short for ***Consumer***, $\boldsymbol{P}$ is short for ***Producer*** and $\boldsymbol{T}$ is short for ***Terminal*** from the example $\boldsymbol{CP}(N_c, N_p)$ model 2.4.3. Source: [24]

2.4.3 Example Consumer/Producer model

Here, we present a working simple Consumer/Produced model modelled in the GPEPA language. This model represents a system of many cooperating ***Consumer*** and ***Producer*** components, where each producer has its own ***Terminal*** and only when the ***Terminal*** is prepared, the ***Producer*** can give out the produced information. It has only one-item buffer, and once in a while, if no ***Consumer*** is interested, clears it. The overall system is then defined as a cooperation between N_c ***Consumer*** and N_p ***Producer*** components.

$$
\begin{aligned}
&\boldsymbol{Consumer} && \stackrel{\mathit{def}}{=} (\mathit{think}, r_t).\boldsymbol{Consumer\_get}\\
&\boldsymbol{Consumer\_get} && \stackrel{\mathit{def}}{=} (\mathit{get\_product}, r_g).\boldsymbol{Consumer\_use}\\
&\boldsymbol{Consumer\_use} && \stackrel{\mathit{def}}{=} (\mathit{use}, r_u).\boldsymbol{Consumer}\\
\\
&\boldsymbol{Terminal} && \stackrel{\mathit{def}}{=} (\mathit{setup}, r_s).\boldsymbol{Terminal}\\
&\boldsymbol{Terminal\_get} && \stackrel{\mathit{def}}{=} (\mathit{get\_product}, \top).\boldsymbol{Terminal}\\
& && + (\mathit{timeout}, r_{ti}).\boldsymbol{Terminal}\\
\\
&\boldsymbol{Producer} && \stackrel{\mathit{def}}{=} (\mathit{init}, r_i).\boldsymbol{Producer\_ready}\\
&\boldsymbol{Producer\_ready} && \stackrel{\mathit{def}}{=} (\mathit{produce}, r_p).\boldsymbol{Producer\_done}\\
&\boldsymbol{Producer\_done} && \stackrel{\mathit{def}}{=} (\mathit{get\_roduct}, r_{gp}).\boldsymbol{Producer}\\
& && + (\mathit{clear}, r_{cl}).\boldsymbol{Producer}
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{CP}(N_c, N_p) \stackrel{\mathit{def}}{=} \mathbf{Consumers}\{\boldsymbol{Consumer}[N_c]\} \underset{\{\mathit{get\_product}\}}{\bowtie}\\
\mathbf{Producers}\{(\boldsymbol{Producer} \underset{\{\mathit{get\_product}\}}{\bowtie} \boldsymbol{Terminal})[N_p]\}
\end{aligned}
$$

2.5 Unified Stochastic Probes and passage-times

Stochastic probes, as introduced in [10] and later extended in [18], are a mechanism for performance querying of stochastic process algebras. They measure probability distributions of how long it takes till a system model executes a certain sequence of actions, often termed as a passage-time density calculation. They are an abstraction independent of all process algebras.

Unified Stochastic Probes are an extension of this concept. They are in detail introduced in [22, Chapter 5] or more concisely in [24, Section III.], content of which we borrowed here. They are passively observing components, which track the progress of a system model. They do it simply by monitoring overall system model emitted actions, or, the extension, actions of a particular component. This was heavily influenced by *Location-aware Probes* [11], but Unified Stochastic Probes improved some of their shortcomings.

We know two basic types of probes. The main ones are called *global probes*. They monitor the actions in the whole system and measure how long it takes until a certain sequence of actions takes place. Apart from observing the system model, they can also observe other probes, referred to as *local probes*. Local probes are attached to a certain particular component forming a component group and monitor its actions only. If they perceive a specified sequence of actions, they can send signals, which other probes, either global, or other local, can catch as actions and advance their own state. Local probes monitoring other local probes are sometimes called *nested local probes*.

Another important property of probes, both local and global, is that they may be repeating. If a probe is repeating, than after observing the sequence of actions it is expecting and sending the signals it is supposed to, it just starts monitoring all over again, expecting the same sequence. It also repeats the signals, when the action sequence expected occurs.

We have been using a word "sequence". The reality is, probes are more flexible than that. We can easily specify a probe, which will happily accept many different sequences of actions. Thanks to using a language inspired by regular expressions, probes are quite easy to learn and use. Before we get to their syntax, we need to show how to actually specify, that we want a probe attached to a system:

$$\begin{aligned} \mathit{ProbedModel} ::= {} & \mathit{Probe}_g \ \textsf{observes} \ \{\mathit{Probe}_l\} \\ & \textsf{where} \ \{\mathit{Location}\} \\ & \textsf{in} \ \ G \\ | \ & \mathit{Probe}_g \ \textsf{in} \ G \end{aligned} \tag{2.6}$$

where Probe_g is the global probe, $\{\mathit{Probe}_l\}$ is a list of local probes, $\{\mathit{Location}\}$ is a list of component substitutions for components cooperating with some probe (or probes) and G is the model we are probing (in our case an iGPEPA model).

2.5.1 Locating a local probe in the system

To better illustrate how $\{\mathit{Location}\}$ are specified with GPEPA, a simple example:

$$\begin{aligned} & \mathbf{Servers}\{\boldsymbol{\mathit{Server}}[?n]\} \Longrightarrow \\ & \qquad \mathbf{Servers}\{(\boldsymbol{\mathit{Server}} \underset{L}{\bowtie} \mathit{Pb}) \wr\wr \boldsymbol{\mathit{Server}}[?n-1]\} \end{aligned}$$

which says, substitute a component group **Servers** in the model with a new component group **Servers**, which is formed by independent cooperation of $n-1$ ***Server*** components and 1 ***Server*** component, which is directly observed by a probe Probe_l on actions in L.

The grammar for this operation is:

$$
\begin{array}{rll}
\mathit{rule} ::= & \mathit{group} \Longrightarrow \mathit{group} & \text{replacement rule} \\
\mathit{group} ::= & H\{\mathit{cpts}\} & \text{component group} \\
\mid & \mathit{group} \underset{\mathit{actions}}{\bowtie} \mathit{group} & \text{group cooperation} \\
\mathit{cpts} ::= & \mathit{cpt} & \text{f-component} \\
\mid & \mathit{cpt} \wr\wr \mathit{cpt} & \text{f-component parallelism} \\
\mid & \mathit{cpt}[\mathit{expr}] & \text{f-component array} \\
\mathit{cpt} ::= & P & i\text{PEPA component} \\
\mid & \mathit{cpt} \underset{\mathit{actions}}{\bowtie} \mathit{cpt} & i\text{PEPA cooperation} \\
\mid & ?p & i\text{PEPA component variable} \\
\mathit{actions} ::= & L & \text{action set} \\
\mid & ?l & \text{action set variable} \\
\mathit{expr} ::= & \mathit{int} & \text{number} \\
\mid & ?n & \text{numeric variable} \\
\mid & \mathit{expr} \oplus \mathit{expr} & \text{binary expression}
\end{array}
$$

where $H \in \mathcal{G}(G)$, $P \in \mathcal{B}(G, H')$ for some $H' \in \mathcal{G}(G)$, $L \subseteq \mathcal{A}_t$, $\mathit{int} \in \mathbb{Z}_+$ and n and p are drawn from a set of variable names.

In our case, using this syntax has the advantage that attaching a local probe to an iGPEPA f-component results again in a standard f-component. Models like these can then be analysed using existing techniques for analysing iGPEPA. Nested local probes also keep this advantage.

2.5.2 Local probes grammar

Now we can advance to the grammar of local probes. A local probe is specified using an actions specification R_l^s and an optional repetition denoted by a $\hookleftarrow$ superscript:

$$\mathit{Probe}_l ::= R_l^s \mid R_l^{s\hookleftarrow} \tag{2.7}$$

where

$$
\begin{array}{rll}
R_l^s ::= & R_l^s, R_l^s & \text{sequence} \\
\mid & R_l : \mathit{signal} & \text{transmitted signal}
\end{array} \tag{2.8}
$$

which formally shows that after observing a specified sequence (or rather specified set of sequences, as this is regular expression based), the local probe will send out a signal.

$$
\begin{aligned}
R_l ::=\ & R_l, R_l && \text{sequence} \\
\mid\ & R_l \mid R_l && \text{choice} \\
\mid\ & R_l ; R_l && \text{both} \\
\mid\ & R_l[n] && \text{iterate } n \text{ times} \\
\mid\ & R_l[m, n] && \text{iterate } m \text{ to } n \text{ times} \\
\mid\ & R_l^? && \text{zero or one} \\
\mid\ & R_l^+ && \text{one or more} \\
\mid\ & R_l^* && \text{zero or more} \\
\mid\ & R_l / R_l && \text{reset} \\
\mid\ & R_l \varnothing R_l && \text{fail} \\
\mid\ & R_l^! && \text{not} \\
\mid\ & . && \text{any action or signal} \\
\mid\ & \mathit{action} && \text{eventual specific action or signal} \\
\mid\ & \overline{\mathit{action}} && \text{subsequent specific action or signal} \\
\mid\ & \epsilon && \text{empty action or signal sequence}
\end{aligned}
\tag{2.9}
$$

where $action \in \mathcal{A}$ is an action (or signal) type.

In performance measurement, we are always interested in the shortest sequence of actions leading to a desired (probe) state. Therefore these expressions are evaluated *lazily* (in the *minimal* manner) - as soon as a sequence satisfying the expression occurs, the expression advances and sends the signal.

The *subsequent specific action* matches the *action* only, whereas the *eventual specific action* is just a shorthand for

$$.^*, \overline{action}, .^*$$

where . is a shorthand for $(\overline{a}_1 \mid \overline{a}_2 \mid \ldots \mid \overline{a}_{|\mathcal{A}(P)|})$ and $a_1 \ldots a_{|\mathcal{A}(P)|} \in \mathcal{A}(P)$, which represents the set of actions in component P to be observed. Informally, . specifies any action.

There are also some operations not present amongst standard regular expressions. "*Both*" construction R_l; R_l matches if an only if both expressions match. The "*reset*" operation R_l/R_l matches when first expression matches. Anytime the second expression matches, matching of the first expression starts all over again. The "*fail*" construction $R_l \varnothing R_l$ is similar to the "reset" construction, but if second expression ever matches, the whole expression fails. Their formal semantics is given in [24, Appendix C].

2.5.3 Global probes grammar

Global probes are similar to the local probes. In the case of a global probe, there is a guarantee, that no other probe observes it. Therefore it makes no sense to define signals in a global probe, thus it is disallowed. However, we still wish to announce start and stop of global measurement and thus we augment the grammar to allow sending two special signals - start and stop. Formally:

$$\begin{aligned} Probe_g ::= {} & R_g : \mathsf{start}, R_g : \mathsf{stop} \\ \mid {} & R_g : \mathsf{start}, R_g : \mathsf{stop}^{\hookleftarrow} \end{aligned} \tag{2.10}$$

The measured passage time is then the time between sending start and stop signals. In the case of a repeating probe, this will give us enough data for steady-state measurement, since the data are identically-distributed. Steady-state, or equilibrium, is when all the system model states have stable probabilities of being active at some time.

R_g is defined as:

$$R_g ::= \{pred\}R_g \mid R_g, R_g \mid \ldots \mid action \mid \overline{action} \mid \epsilon \tag{2.11}$$

Apart from $\{pred\}R_g$ rule this is identical to the R_l grammar.

The rule $\{pred\}R_g$ allows to start matching R_g only if the state guard predicate $pred$, a boolean expression, is true. The syntax of predicates:

$$\begin{aligned}
pred ::= {} & \mathsf{true} \mid \mathsf{false} && \text{boolean} \\
\mid {} & \neg pred && \text{not} \\
\mid {} & pred \vee pred && \text{disjunction} \\
\mid {} & b\_expr && \text{expression} \\
b\_expr ::= {} & r\_expr \succeq r\_expr && \text{comparison} \\
r\_expr ::= {} & H : P && \text{component count} \\
\mid {} & int && \text{number} \\
\mid {} & r\_expr \oplus r\_expr && \text{arithmetic} \\
\succeq ::= {} & = \mid \geq \mid \leq && \text{relational ops} \\
\oplus ::= {} & + \mid - && \text{binary ops}
\end{aligned} \tag{2.12}$$

where, if G is the iGPEPA model to which the global probe is to be attached, $(H, P) \in \mathcal{B}(G)$.

The formal semantics for the predicate language for a iGPEPA model in a state s is given by:

$$\begin{array}{ll}
s \models \mathsf{true} & \text{for all } s \\
s \models \mathsf{false} & \text{for no } s \\
s \models \neg\psi & \mathsf{iff}\ s \not\models \psi \\
s \models \psi_1 \vee \psi_2 & \mathsf{iff}\ s \models \psi_1 \vee s \models \psi_2 \\
s \models b\_expr & eval(b\_expr, s)
\end{array}$$

where b_expr is, as above, a boolean function of $H_1 : P_1, H_2 : P_2, \ldots$ expressions, which reference specific f-component derivative states P_x in the component groups H_x. The $eval$ function evaluates the boolean function by dereferencing the $H_x : P_x$ expressions as the number of P_x components active in the component group H_x in the particular state of the model s.

2.5.4 Calculating passage time densities using the Unified Stochastic Probes

The Unified Stochastic Probes were specifically designed to measure passage-time densities, e.g. the time it takes to transfer from a component state S to a component state T, or the time it takes a certain sequence of actions to occur, when in state S. There are number of passage-time analyses used, depending on the circumstances. We can observe passages while the system is in the steady-state, or we can observe the system from certain time t. The former analysis is often termed as the *steady-state passage time*, whereas the latter as the *transient passage time* analysis. The steady-state passage time analysis is a special case of the transient passage time analysis, with the time t chosen to be approaching the infinity.

We also distinguish between observing the system as a whole and observation of its individual components. When we observe one individual component, we are interested in the *individual* passage time of this component cooperating with a large number of identical components. It is not important, which particular component from these we observe, since they are identical and probabilistically, they will expose the same behaviour. We can also monitor multiple (not necessarily identical) components. In this case we are interested in the *global* passage time. Location-aware probes simplify this process, since we can attach them to a particular component in its arbitrary state. If we do not use probes, we can achieve the same using the GPEPA models [25].

Steady-state individual passage time

To specify a probe observing a steady-state individual passage time, we use the following form:

$$\begin{aligned}
\boldsymbol{PM} &\overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\quad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} R_l : \mathsf{begin}, R_l : \mathsf{end}^{\hookleftarrow} \\
&\quad \mathsf{where} \quad \mathbf{H}\{\boldsymbol{P}[?n]\} \Longrightarrow \\
&\qquad \mathbf{H}\{(\boldsymbol{P} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{P}[?n-1]\} \\
&\quad \mathsf{in} \quad \boldsymbol{G}
\end{aligned}$$

This form formally requests, that we measure the system with a repeating local probe attached to the component of interest, $\boldsymbol{P}$. We then leave the model running until a time t approaching the infinity. At this time, the model should be stable and have stable state probabilities. Only then we substitute the local probe for another, non-repeating (absorbing) one. Formally, we consider our model $\boldsymbol{G}$ at the time t and start running a new model $\boldsymbol{G}'$ with the same initial component counts as were the component counts of $\boldsymbol{G}$ at time t. The component $\boldsymbol{P}$ is attached to a non-reapeating version of the local probe $Probe_l$, called $Probe_l{}'$, and we leave the $\boldsymbol{G}'$ running until the $Probe_l{}'$ is in an accepting state. We can repeat this simulation many times over to obtain the frequency distribution of the finishing times. After normalising, this is our passage time density, which is then easy to convert to a cumulative distribution function (CDF).

There is one small catch. For certain probes, the state space of repeating and non-repeating probes combined with the model might differ considerably. If we ignore one extra state in non-repeating version, which causes no trouble, since we just expect the simulation to reach this state in second run, it might be the case, that some combined states are reachable with repeating probe, but not reachable with its non-repeating version. This may cause issues when assigning from $\boldsymbol{G}$ to $\boldsymbol{G}'$, since at that particular moment, the probed component in $\boldsymbol{G}$ may be in one of these states.

One way to solve this problem is to wait enough time after the begin signal, when the probe reaches a combined state common to both models. This approach could work, if we generated all the possible states for both models and all the possible events leading from each of them. This is quite computationally inefficient.

Another approach is to choose the state after the begin signal probabilistically. We assign weights to the each state representing the probability of being there right after the begin signal. We generate a random number from uniform distribution between 0 and 1.[1] Then we simply consider our combined states in an ordering we chose beforehand, and cumulate their weights. The first one in the order, which has the cumulative weight equal or larger to our generated number, will be chosen. The question here is how to assign the weights. We can borrow a formula from [24], where a similar approach was taken but for fluid flow approximation.

$$\frac{\sum_{\alpha\in\mathcal{A}_t, C\in ds^*(P \underset{*}{\bowtie} Pb), C \xrightarrow{(\alpha,\lambda),(\ldots,\mathsf{begin},\ldots)} Q} \left(\frac{\mathcal{R}_\alpha(G,V,H,C)\lambda}{r_\alpha(C)}\right)}{\sum_{\alpha\in\mathcal{A}_t, C\in ds^*(P \underset{*}{\bowtie} Pb), C \xrightarrow{(\alpha,\lambda),(\ldots,\mathsf{begin},\ldots)}} \left(\frac{\mathcal{R}_\alpha(G,V,H,C)\lambda}{r_\alpha(C)}\right)} \tag{2.13}$$

where $V \in \mathcal{B}(G) \to \mathbb{Z}_+$ is defined by $\tilde{V}(Y,Q) := v_{Y,Q}$ for all $(Y,Q) \in \mathcal{B}(G)$.

The probability of a state $\boldsymbol{Q}$ being active right after the begin signal has been fired is the sum of the expected rates of begin transitions leading to $\boldsymbol{Q}$ divided by the sum of all expected rates of begin transitions. To better understand the inner term, imagine that as the rate of the transition given our model multiplied by the probability of this action occurring in the component $\boldsymbol{C}$. This is the approach we adopted.

Transient individual passage time

For transient individual passage time, we require similar form to the steady-state individual passage time; however, the local probe is non-repeating. In case of probes, we can consider the starting time t of transient individual passage time analysis to be the time, when the monitored local probe sends the begin signal (or equivalently the global probe sends the start signal).

[1]In a uniform distribution, each number in range has the same probability of being chosen.

$$\begin{aligned}
&\boldsymbol{PM} \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop} \\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} R_l : \mathsf{begin}, R_l : \mathsf{end} \\
&\qquad \mathsf{where} \quad \mathbf{H}\{\boldsymbol{P}[?n]\} \Longrightarrow \\
&\qquad\qquad \mathbf{H}\{(\boldsymbol{P} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{P}[?n-1]\} \\
&\qquad \mathsf{in} \quad \boldsymbol{G}
\end{aligned}$$

This is evaluated by simply attaching the $Probe_l$ to the $\boldsymbol{G}$ and running the model, until the $Probe_l$ is in an accepting state. By repeating the experiment, we can again obtain the frequency distribution of the times when this happens and after normalising our passage time density.

Global passage times

These leave much more freedom for specification. The user can use the full global probes grammar with predicates, and optionally arbitrary local probes and substitutions. Formally, the form is allowed to be as follows:

$$\begin{aligned}
&\boldsymbol{PM} \overset{def}{=} \epsilon : \mathsf{start}, R_g : \mathsf{stop} \\
&\qquad \mathsf{observes} \quad \{Probe_l\} \\
&\qquad \mathsf{where} \quad \{Location\} \\
&\qquad \mathsf{in} \quad G
\end{aligned}$$

The evaluation consists of applying the substitutions in the order of specification. Next step is to attach the global probe to the model, run it and obtain the time when the global probe sends the stop signal. Once again, we obtain the passage time density by obtaining the frequency distribution of measurement completion times. We should note that currently, simulation does not support predicates. This is still an open research question, as there are multiple ways how to achieve its support. They require some extension of PEPA, for example to support functional rates (rates dependent on some factor rather than constant rates).

2.5.5 Examples

In this section we present some simple examples of probes. We are interested in particular kinds of probes for determining individual or global passage time distributions. For individual passages of a single component, we either determine the distribution from certain time t (transient individual) or in equilibrium (steady-state individual). These probes will be attached to the GPEPA example model from the Subsection 2.4.3.

In such a system, it would make sense to specify an SLA. For instance, one that guarantees that any ***Consumer*** will at least in 99% of time use minimum of two pieces of information within $250ms$ after performing *think* action for the first time. This can be expressed by an individual steady state passage time calculated by a global probe observing a local probe attached to a single ***Consumer*** component. Asterisk as cooperation actions set stands for all the model actions monitored by $Probe_l$, in this case *think* and *use*.

$$
\begin{aligned}
&\boldsymbol{PM}_1 \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} think : \mathsf{begin}, use[2] : \mathsf{end}^{\hookleftarrow} \\
&\qquad \mathsf{where} \quad \mathbf{Consumers}\{\boldsymbol{Consumer}[N_c]\} \Longrightarrow \\
&\qquad\qquad \mathbf{Consumers}\{(\boldsymbol{Consumer} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{Consumer}[N_c - 1]\} \\
&\qquad \mathsf{in} \quad \boldsymbol{CP}(N_c, N_p)
\end{aligned}
$$

Or we might be interested in the time it takes one ***Producer*** to empty its buffer. This can be observed with a transient individual passage time probe:

$$
\begin{aligned}
&\boldsymbol{PM}_2 \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop} \\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} \epsilon : \mathsf{begin}, clear : \mathsf{end} \\
&\qquad \mathsf{where} \quad \mathbf{Producers}\{(\boldsymbol{Producer} \underset{get\_product}{\bowtie} \boldsymbol{Terminal})[N_p]\} \Longrightarrow \\
&\qquad\qquad \mathbf{Producers}\{((\boldsymbol{Producer} \underset{get\_product}{\bowtie} \boldsymbol{Terminal}) \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \\
&\qquad\qquad\quad (\boldsymbol{Producer} \underset{get\_product}{\bowtie} \boldsymbol{Terminal})[N_p - 1]\} \\
&\qquad \mathsf{in} \quad \boldsymbol{CP}(N_c, N_p)
\end{aligned}
$$

We might also be interested in the overall system behaviour, for example how long will it take at least 30% of ***Consumer*** components to obtain at least 10 products. For this we can use a global passage time probe:

$$
\begin{aligned}
&\boldsymbol{PM}_3 \overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_c * 0.3] : \mathsf{stop} \\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} get\_product[10] : \mathsf{end} \\
&\qquad \mathsf{where} \quad \mathbf{Consumers}\{\boldsymbol{Consumer}[N_c]\} \Longrightarrow \\
&\qquad\qquad \mathbf{Consumers}\{(\boldsymbol{Consumer} \underset{*}{\bowtie} \boldsymbol{Probe}_l)[N_c]\} \\
&\qquad \mathsf{in} \quad \boldsymbol{CP}(N_c, N_p)
\end{aligned}
$$

Please note the rather clumsy and complicated description in English even for so simple probes and compare it to the clear formal syntax of Unified Stochastic Probes.

2.6 Unified Stochastic Probes translation to iGPEPA

The process of translating the Unified Stochastic Probes to iGPEPA is straightforward. It consists of only few steps, which we will introduce below, details in [24].

The usual practice when working with regular expression is their conversion to a *non-deterministic finite automata* (NFA). An NFA is a simple automata graph, which is characteristic by being non-deterministic. A usual NFA graph has plenty of so-called empty transitions, which do nothing else than just move the current state, where different transitions may be available. Since they are empty, they do not match anything. The execution could follow them, but it does not have to. A simple example graph is shown in the Figure 2.2.

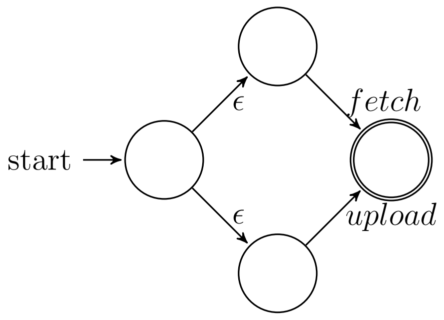

Figure 2.2: NFA example

In this graph, we can take either ϵ and then match $fetch$ or *upload* respectively. We cannot match both, after we take any of the ϵs. That is exactly the disadvantage of the NFAs - to match with them, we need a fair amount of backtracking (returning to states we have already visited), which adds computational complexity. Furthermore, PEPA has no equivalent to an empty transition. Thus we first need to convert the NFA to a *deterministic finite automata* (DFA).

Since probes syntax is based on regular expressions, it is easy to find a literature on how to translate their R_l to NFAs, with the exception of the non-standard operations - "both", "reset" and "fail". These are better described in the implementation chapter (Subsection 4.6.2) and formally in [24, Appendix C]. To translate R_l^s, we do the following steps:

1. Translate R_l.

2. Add a new state $SigState$ with a transition named after the signal s from R_l^s, which leads to a new accepting state A.

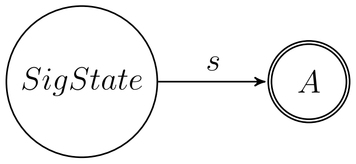

Figure 2.3: Signal transition from $SigState$

3. All the accepting states in R_l are changed to non-accepting and given an empty transition to $SigState$.

If there are multiple R_l^s, all of them are joint together - the starting state of the next R_l^s is linked with the accepting state of the current R_l^s by an empty transition. This accepting state is then made non-accepting.

The next step is to convert the complete NFA to a DFA. DFA is a graph, which contains no non-deterministic decisions and no empty transitions. They are very fast to use for pattern matching. Their disadvantage is, that in general, we need much more states for a DFA than for an equivalent NFA. In the worst case, when converting from NFA, we need 2^n states, where n is the number of the states in the original NFA.

There is one more difference between DFA and NFA. DFAs have an explicit failure state. If they are matched against a pattern, which has no available transition in the current state, we end up in the failure state. Once the failure state is entered, it cannot be left (it has the self-loops for all the possible transitions). However, for our purposes, we do not need a failure state and we will ignore it.

The conversion from NFA to DFA is a well-known problem and we will not describe it in detail. For interested to read further, a good source is [19]. After conversion, the DFAs are usually quite clumsy. Another well-known algorithm, also described in the given link, is then minimisation (optimisation) of the DFA. It can be shown, that for each pattern matched by a DFA, there exists a DFA m with a minimum number of states. There may be many more DFAs correct for this pattern, but they can all be reduced to m.

Next we make the translation of our local probes to iGPEPA. This step is very simple. Each state is translated to one component state. All these component states belong to the same compponent. The transitions are then translated as the available choices with passive rates. If there is a signal transition, then instead of a passive rate it will become an immediate action. For each state, if there is no transition corresponding to an action from cooperation between the probe and the observed component, we will add a self-loop passive prefix with this action. If the probe is non-repeating, the accepting state will have all its actions as self-loops.

Below we present a simple example from Subsection 2.5.5. The local probe would be translated to an NFA as in the Figure 2.4.

$$\begin{aligned}
&\boldsymbol{PM}_1 \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} \mathit{think} : \mathsf{begin}, \mathit{use}[2] : \mathsf{end}^{\hookleftarrow} \\
&\qquad \mathsf{where} \quad \mathbf{Consumers}\{\boldsymbol{Consumer}[N_c]\} \Longrightarrow \\
&\qquad\qquad \mathbf{Consumers}\{(\boldsymbol{Consumer} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr \boldsymbol{Consumer}[N_c - 1]\} \\
&\qquad \mathsf{in} \quad \boldsymbol{CP}(N_c, N_p)
\end{aligned}$$

The global probe cannot be fully translated to iGPEPA because of the predicates. For the simulation purposes, we impose a restriction on grammar to not contain any predicates. For fluid flow approximation (described in the Subsection 2.7.3), we do not translate them at all.

$$\begin{tikzcd}[row sep=large]
Probe_l \arrow[d, "think"] \\
Probe_l2 \arrow[d, "\mathsf{begin}"] \\
Probe_l3 \arrow[d, "use"] \\
Probe_l4 \arrow[d, "use"] \\
Probe_l5 \arrow[d, "\mathsf{end}"] \\
Probe_l6 \arrow[uuuuu, bend left=40, "\epsilon"]
\end{tikzcd}$$

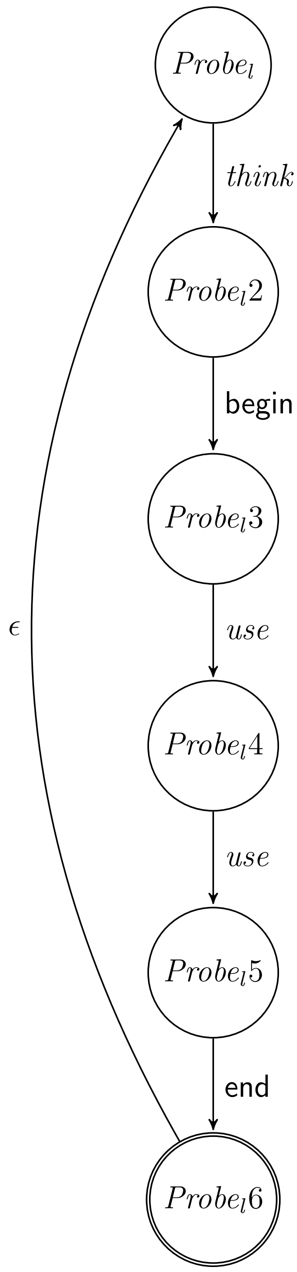

Figure 2.4: Signal transition from *SigState*

2.7 Fluid-flow approximation of PEPA models

As we have previously stated, PEPA models are easily translatable to CTMCs. We know some methods for solving these and therefore we are able to state our performance queries. However, translating more complicated models directly usually causes a problem called state-space explosion. This essentially means that the resulting CTMC has too many states to be computationally tractable. Even without this problem though, solving CTMCs is computationally a very demanding process. Improving the performance of CTMC solving has thus been a topic for recent research. Although there have been some advancements, it is still a slow process. We do not talk about CTMCs in this work, but we would like to point out [13], which introduces the subject.

A new approximation method for PEPA models was proposed in [27]. It is called *fluid-flow approximation* and it avoids the direct translation of the model to a CTMC. Instead, it first does a state-space aggregation of the model states and after that uses ordinary differential equations to represent the overall model behaviour which can be solved in incomparably shorter time than simulation would take.

2.7.1 State-Space Aggregation

When we consider a PEPA model, it is straightforward to translate it to a graph. Each component state will represent a node and each action with associated rate is represented as an arc. For example, if we have $\boldsymbol{Client_0} \overset{def}{=} (think, r_t).\boldsymbol{Client_1}$, we could translate it as in the Figure 2.5.

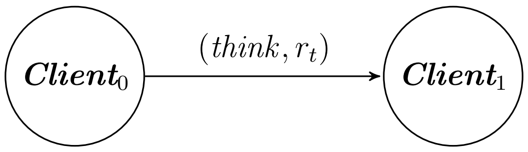

Figure 2.5: PEPA as graph, $\boldsymbol{Client_0} \overset{def}{=} (think, r_t).\boldsymbol{Client_1}$

When two components are cooperating, a simple way of making a graph then is to consider the cooperation as one state. When an unsynchronized action occurs, left or right-hand side will evolve. If synchronized, than both. A little example:

$$\begin{aligned} \boldsymbol{Client_0} &\overset{def}{=} (think, r_t).\boldsymbol{Client_1} & \qquad \boldsymbol{Server_0} &\overset{def}{=} (initialise, r_i).\boldsymbol{Server_1} \\ \boldsymbol{Client_1} &\overset{def}{=} (request, r_{r1}).\boldsymbol{Client_2} & \qquad \boldsymbol{Server_1} &\overset{def}{=} (request, r_{r2}).\boldsymbol{Server_2} \end{aligned}$$

$$\boldsymbol{SC}(1,1) \overset{def}{=} \text{Clients}\{\boldsymbol{Client_0}\} \underset{\{request\}}{\bowtie} \text{Servers}\{\boldsymbol{Server_0}\}$$

Let's suppose the following actions occurred: *think*, *initialise*, *request*.

Then the model will go through these states:

$$\text{Client}\{\boldsymbol{Clients}_0\} \underset{\{request\}}{\bowtie} \text{Servers}\{\boldsymbol{Server}_0\}$$
$$\text{Client}\{\boldsymbol{Clients}_1\} \underset{\{request\}}{\bowtie} \text{Servers}\{\boldsymbol{Server}_0\}$$
$$\text{Client}\{\boldsymbol{Clients}_1\} \underset{\{request\}}{\bowtie} \text{Servers}\{\boldsymbol{Server}_1\}$$
$$\text{Client}\{\boldsymbol{Clients}_2\} \underset{\{request\}}{\bowtie} \text{Servers}\{\boldsymbol{Server}_2\}$$

For two components that is not such a problem. Imagine though, we have 100 $\boldsymbol{Client}_0$ and 60 $\boldsymbol{Server}_0$ components. The number of required states would grow exponentially. Fortunately, this can be easily avoided. Instead of considering each component individually, we can aggregate the components of the same type and represent them by one number in the vector of component counts. For the example mentioned, the initial state would be $(100, 0, 0, 60, 0, 0)$, which says, there are 100 $\boldsymbol{Client}_0$ components, 0 $\boldsymbol{Client}_1$, 0 $\boldsymbol{Client}_2$, 60 $\boldsymbol{Server}_0$, 0 $\boldsymbol{Server}_1$ and 0 $\boldsymbol{Server}_2$. After executing our example actions, the vector would develop in this way: $(99, 1, 0, 60, 0, 0)$, $(99, 1, 0, 59, 1, 0)$ and $(99, 0, 1, 59, 0, 1)$. Simply put, if there is an action performed in an aggregated group of components in the same state, one of the entries in the component counts vector will be increased, one decreased and these entries may be the same one. If this is a shared action, this will happen to each aggregated group participating in the action.

This simple change of thought drastically decreases the state space for usual PEPA models. In addition, it appears there is no significant information loss, since the replicated components are identical and thus we can disregard the knowledge about the particular component, which evolved. With aggregated space we could already create a CTMC (called PCTMC, Population CTMC) with much less complexity and try to solve it instead of the original one. However, we can go even further.

2.7.2 Fluid-flow approximation

As presented in [27, Section 3], we can avoid the CTMC computation altogether. Instead, we can derive a set of ordinary differential equations, which will describe the behaviour of the system. They will allow us to regard a system as a set of count vectors at certain times.

Conversion to PCTMC (or CTMC) is an optional step. It allows to use PCTMC as an abstraction of process algebras (PEPA, Stochastic Petri Nets, etc.), as long as we can convert these to a PCTMC. This is the approach GPAnalyser [29] has adopted. As discussed in the Section 2.3, we do not describe CTMCs in this work.

The component count vector presented in the previous section is *discrete* and its entries are always increased or decreased by one. However, with a large number of components, we can accept the assumption, that the vector is *continuous* and its entries can change by a very small amount. In that case we are able to approximate these values at all times.

First, some preliminary definitions. An activity a is an *exit* activity for component $\boldsymbol{C}$, if the activity is enabled by $\boldsymbol{C}$, in other words if there is a transition of type $\boldsymbol{C} \xrightarrow{(a,r)}$. *Entry* activity, if there is a transition of type $\xrightarrow{(a,r)} \boldsymbol{C}$. For each of the components and their states we will define their sets of entry (*entering*) and exit activities. We will also define the entry and exit sets for each action, which will contain all the component states, for which this action is an entry or exit action respectively.

Now we are prepared for the technique itself. Change of number of components being in some component state s during an $[t, \delta t]$ interval can be interpreted as the difference of components entering and leaving the state s during this period. Formally, we can write a formula:

$$\begin{aligned} N(\boldsymbol{C}_{i_j}, t+\delta t) - N(\boldsymbol{C}_{i_j}, t) = \\ - \sum_{(a,r_i)\in Ex(\boldsymbol{C}_{i_j})} \min_{\boldsymbol{C}_{k_l}\in Ex(a,r)} (r \times N(\boldsymbol{C}_{k_l}, t))\delta t \\ + \sum_{(a,r_i)\in En(\boldsymbol{C}_{i_j})} \min_{\boldsymbol{C}_{k_l}\in Ex(a,r)} (r \times N(\boldsymbol{C}_{k_l}, t))\delta t \end{aligned}$$

where C_{i_j} represents the $i-th$ component type in the system and its $j-th$ derivative state, and $N(C_{i_j}, t)$ the number of C_{i_j} in the system at time t.

The second line represents the decrease of the number of $\boldsymbol{C}_{i_j}$ by pursuing their exit activities. If a is an unsynchronized activity of this component state then $\min_{C_{k_l}\in Ex(a,r)}(N(C_{k_l}, t))$ will be simply $N(C_{i_j})$. Naturally, there may be another component, for which a is an exit activity, however, since there is no synchronisation, we consider it to be in a different scope. According to the definition of apparent rate, n replicated components $\boldsymbol{C}$ will execute (d, r) with the apparent rate $n \times r$.

By the semantics of the apparent rate, cooperating components execute a synchronized action with the apparent rate of the slower one. If $\boldsymbol{C}_{i_j}$ shares this activity, we will take the cooperating party with the lowest apparent rate as the new apparent rate.

Analogically to that, we can explain the third line of the equation representing the impact of entry activities, which are represented as the exit activities of other component states entering this component state.

With this equation, we can continuously compute this change for any component state in the system and for any two times. However, dividing the whole equation by δt and by taking the limit with δt converging to 0 we obtain:

$$\begin{aligned} \frac{dN(\boldsymbol{C}_{i_j}, t)}{dt} = - \sum_{(a,r_i)\in Ex(\boldsymbol{C}_{i_j})} \min_{\boldsymbol{C}_{k_l}\in Ex(a,r)} (r \times N(\boldsymbol{C}_{k_l}, t)) \\ + \sum_{(a,r_i)\in En(\boldsymbol{C}_{i_j})} \min_{\boldsymbol{C}_{k_l}\in Ex(a,r)} (r \times N(\boldsymbol{C}_{k_l}, t)) \end{aligned}$$

We can see this ordinary differential equation represents the evolution of the system for a component state at certain time. This is the key technique to approximating the development of the system model. Experiments, as provided in [27] show that for large numbers of replicated components, we can obtain results very similar to the CTMC numerical solving or the plain simulation, which has been formally proved in [21].

As shown in [23], this concept can be easily extended to GPEPA models. We can define

$v_{H,P}(t)$ to be the real-valued fluid flow approximation of $N_{H,P}(t)$ in GPEPA and also V_t to be a function defined as $V_t(H,P) = v_{H,P}(t)$. Then we can borrow the definition of the component rate function.

Definition 2.1 (Component rate)**.** *Let G be a* i*GPEPA model. For $(H,P) \in \mathcal{B}(G)$, timed action type $\alpha \in \mathcal{A}_t$ and $V_t \in \mathcal{B}(G) \to \mathbb{R}_+$, the component rate function is defined as follows.*

$$\mathcal{R}_\alpha(M_1 \underset{L}{\bowtie} M_2, V_t, H, P) := \begin{cases} \frac{\mathcal{R}_\alpha(M_i,V_t,H,P)}{r_\alpha(M_i,V_t)} \min(r_\alpha(M_1,V_t), r_\alpha(M_2,V_t)) \\ \qquad \textit{if } \alpha \in L \textit{ and } H \in \mathcal{G}(M_i), \textit{ for } i = 1 \textit{ or } 2 \\ \mathcal{R}_\alpha(M_i, V_t, H, P) \\ \qquad \textit{if } \alpha \notin L \textit{ and } H \in \mathcal{G}(M_i), \textit{ for } i = 1 \textit{ or } 2 \end{cases}$$

$$\mathcal{R}_\alpha(Y\{D\}, N, H, P) := \begin{cases} V_t(H,P)\, r_\alpha(P) & \textit{if } H = Y \textit{ and } P \in \mathcal{B}(G,H) \\ 0 & \textit{otherwise} \end{cases}$$

Terms with zero-valued denominators are defined to be zero.

Intuitively, it specifies the rate of an action occurring in a GPEPA component, or set of components in relation to the rest of the model. For cooperation between two GPEPA groups, we either have unchanged rate for each of them for an action α, if they do not synchronise on it, or the rate is limited by the slower component. For a single component type, its rate of action α is given by the number of such components at that particular time, which is expressed by the second part of the definition. This follows from the superposition property of a Poisson process, on which PEPA actions are based.

This function is specified in terms of the count-oriented apparent rate function for GPEPA models.

Definition 2.2 (Count-oriented apparent rate)**.** *Let G be an* i*GPEPA model. Let $\alpha \in \mathcal{A}_t$ be a timed action type and $V_t \in \mathcal{B}(G) \to \mathbb{R}_+$. Then the apparent rate is defined as follows.*

$$r_\alpha(M_1 \underset{L}{\bowtie} M_2, V_t) := \begin{cases} \min(r_\alpha(M_1,V_t), r_\alpha(M_2,V_t)) & \textit{if } \alpha \in L \\ r_\alpha(M_1,V_t) + r_\alpha(M_2,V_t) & \textit{otherwise} \end{cases}$$

$$r_\alpha(Y\{D\}, V_t) := \sum_{P \in \mathcal{B}(Y\{D\},Y)} V_t(Y,P)\, r_\alpha(P)$$

Informally, this function expresses similar ideas as the component rate function, however it deals with the apparent rate of an action in this subsystem only - the rate with which the action would occur if we ignored the rest of the system.

We also define the derivative weighting function, which expresses the probability, that from a state $\boldsymbol{P}$ we travel to state $\boldsymbol{Q}$.

Definition 2.3 (Derivative weighting function)**.** *Let G be an iGPEPA model and $H \in \mathcal{G}(G)$ a component group label. Let $P, Q \in \mathcal{B}(G,H)$ be f-component derivative states and let $\alpha \in \mathcal{A}_t$*

be a timed action type. Then $p_\alpha(P,Q) := \frac{1}{r_\alpha(P)} \sum_{P \xrightarrow{(\alpha,\lambda),\cdot} Q} \lambda$*. This is defined to be zero when* $r_\alpha(P) = 0$*.*

With these definitions, we can simplify the equation we obtained to

$$\dot{v}_{H,P}(t) = \underbrace{\sum_{\alpha \in \mathcal{A}_t} \left(\sum_{Q \in \mathcal{B}(G,H)} p_\alpha(Q,P)\, \mathcal{R}_\alpha(G, V_t, H, Q) \right)}_{(1)} - \underbrace{\sum_{\alpha \in \mathcal{A}_t} \mathcal{R}_\alpha(G, V_t, H, P)}_{(2)} \tag{2.14}$$

which, although not really expressing new ideas, simplifies the notation and generalizes it to GPEPA models.

As shown in [24, Section IV.], we can also estimate the number of times a certain action a has occurred:

$$\dot{v}_a(t) = r_a(G, V_t)$$

Intuitively, this says the number of action a occurred will increase with the statistical rate of them happening in the system.

2.7.3 Approximating the passage time probabilities

With fluid-flow approximation, we can already query the system for the number of components or executed actions at a certain time. We can go even further though. The approximated component counts can serve as the basis for the passage time calculations.

As discussed in Subsection 2.5.4, with the Unified Stochastic Probes we can easily measure steady-state individual, transient individual and global passage times. Rather than using the slow simulation, we can use the obtained component counts for approximation of the passage time densities. This is thoroughly demonstrated in [22, Chapter 5]. For our needs, we only discuss it briefly.

Steady-state individual passage time approximation

In this mode we run the model similarly to the approach described in the Subsection 2.5.4. Given a probe for steady-state passage time, we first attach the repeating local probe to the model $\boldsymbol{G}$ according to the substitution. We then run the model fluid flow approximation for the time approaching the infinity. Afterwards, as in simulation, we initialise the model $\boldsymbol{G'}$ with the initial counts of components obtained from this run. Again, in $\boldsymbol{G'}$ we use the non-repeating (absorbing) version of the local probe. In the case of fluid flow approximation, it is perfectly acceptable for the counts to be non-integer numbers.

The tricky part is which state we should initialise the observed component to. We are essentially looking for a state, where the component will be immediately after firing the begin signal. For simple models, this might be one particular state reached by immediate begin transition. In general however, the more complicated probes in synchronization with our component will have multiple states, to which begin signal may lead. Since V_t is a real numbered function, we can directly assign the probabilities of each state to be active at this stage to it. The probability of a state $\boldsymbol{Q}$ being active right after the begin signal has been fired is the sum of the expected rates of begin transitions leading to $\boldsymbol{Q}$ divided by the sum of all expected rates of begin transitions. Formally:

$$\frac{\sum_{\alpha\in\mathcal{A}_t, C\in ds^*(P \underset{*}{\bowtie} Pb), C \xrightarrow{(\alpha,\lambda),(\dots,\mathsf{begin},\dots)} Q} \left(\frac{\mathcal{R}_\alpha(G,V,H,C)\lambda}{r_\alpha(C)}\right)}{\sum_{\alpha\in\mathcal{A}_t, C\in ds^*(P \underset{*}{\bowtie} Pb), C \xrightarrow{(\alpha,\lambda),(\dots,\mathsf{begin},\dots)}} \left(\frac{\mathcal{R}_\alpha(G,V,H,C)\lambda}{r_\alpha(C)}\right)} \tag{2.15}$$

where $V \in \mathcal{B}(G) \to \mathbb{Z}_+$ is defined by $\tilde{V}(Y,Q) := v_{Y,Q}$ for all $(Y,Q) \in \mathcal{B}(G)$.

To better understand the inner term, imagine that as the rate of the transition given our model multiplied by the probability of this action occurring in the component $\boldsymbol{C}$. This function then defines the initial setup for the probed component $\boldsymbol{P}$ and is directly assigned to $v'_{H,P}(0)$ for our second model. For all the unprobed components, as mentioned above, we will simply use $v'_{H,C}(0) = v_{H,C}(t)$.

After we run the second model $\boldsymbol{G'}$ for a given amount of time, we can inspect the model at each of the time instances. For each of these we define the passage time density simply as the probability that that observed component is in the probe accepting state, as the probe is non-repeating. Formally:

$$F_{\mathrm{si}}(t) := \sum_{C \in - \underset{*}{\bowtie} Pb_{\mathrm{Acc}}} v'_{H,C}(t) \tag{2.16}$$

where Pb_{Acc} is an accepting state of the local probe.

Transient individual passage time approximation

Transient analysis is similar to the steady-state analysis from the previous section. This time, we only need one model, since the probe is already non-repeating. We can define the formula to compute a CDF of density of reaching the accepting state of local probe given that we started measuring at time s (i.e. the begin was sent at time s).

$$F^s_{\mathrm{ti}}(t) := \sum_{C \in - \underset{*}{\bowtie} Pb_{\mathrm{Acc}}} v^s_{H,C}(t)$$

This is principally identical to the formula from the previous section (2.16). We can then obtain the unconditional CDF with the following formula:

$$F_{\mathrm{ti}}(t) := \int_0^\infty F^s_{\mathrm{ti}}(t) \times \frac{\mathrm{d}K(s)}{\mathrm{d}s}\,\mathrm{d}s$$

where $K(s) := \sum_{Q \in \mathcal{Q}} v_{H,Q}(s)$ symbolizes the probability of being in a state, which was reached after firing the begin signal at time s, i.e. $\mathcal{Q}$ is the set of all states reachable after emitting the begin signal. We use its derivative, representing the probability that begin was signalled at time instant s, for weighing the CDFs. The overall $K(s)$ will eventually converge to 1, since we have exactly one probed component. When it does so, the instantaneous probability, its derivative, will converge to zero. That means we can, with some acceptable error, truncate the integral computation for shorter time than infinity, usually in magnitude of hundreds or thousands units, although this heavily depends on the model.

Lastly, we need to define the initial conditions $v^s_{H,C}(0)$ for each component $\boldsymbol{C}$ for each time s. Since these represent the model state given the begin was emitted at time s, we can reuse the formula from the previous section (2.15). For unprobed components we will simply use the $v^s_{H,C}(0) = v_{H,C}(s)$ assignment. For the probed component $\boldsymbol{P}$, we will use the formula in the same manner.

For local probes of the form ϵ : begin, R_l : end the begin signal is fired immediately and therefore the formula can have some other value than 0 at $v'_{H,P}(0)$, which is $v_{H,P}(0)$. With local probes defined in this way, we know that the begin signal will be always fired immediately after starting the model. Therefore we can simplify the overall computation and compute just $F^0_{\mathrm{ti}}(t)$ as the unconditional CDF.

Global passage time approximation

Global passage time can provide us with point-mass-approximation of the passage time. We should note, that user is free to specify arbitrary local probes and their substitutions, as for the simulation. Sadly, for the global probes, only a limited set of operations is currently permitted, since there has not been a formal proof for the rest of the operations. This grammar is specified as:

$$\begin{aligned} R_g ::= \;& \{pred\}R_g \;\mid\; R_g, R_g \;\mid\; R_g \mid R_g \\ & \mid\; R_g; R_g \;\mid\; R^a_g[n] \end{aligned} \tag{2.17}$$

where:

$$R^a_g ::= R^a_g \mid R^a_g \;\mid\; action$$

and $action \in \mathcal{A}$ is an action (or signal) type.

Informally, we are able to measure predicated probes (cf. to simulation), and some basic operations - "sequence", "both" and "choose" on operands. Operands are either another of these operations, or, in R^a_g case, n occurrences of actions from the specified set of actions.

The way to evaluate this is then to apply all the substitutions in the order of appearance. Then, with a model $\boldsymbol{G'}$ changed in such way, we can run the fluid flow approximation. At all the times, we look at the component and executed action counts and evaluate, if we should progress our state. In this mode, unlike in the simulation mode, we cannot translate the global probe to the iGPEPA. That would mean cooperating on immediate actions between the groups, which would lead to ill-formed differential equations system, as discussed in [22]. Instead we can formally define a function $\mathcal{U}(R, e)$. It is mapped to the time, when the expression R will finish evaluation if starting in time e. Since we are interested in the overall execution of the global probe, the time of interest is defined as $\mathcal{U}(R, 0)$, where R is the sequence of R_g expressions for `start` and `stop` signals. It is defined as follows:

$$\begin{aligned}
\mathcal{U}((R_1, R_2), e) &:= \mathcal{U}(R_2, \mathcal{U}(R_1, e)) && \text{R_2 starts when R_1 finishes} \\
\mathcal{U}((R_1 \mid R_2), e) &:= \min(\mathcal{U}(R_1, e), \mathcal{U}(R_2, e)) && \text{minimal time of finish preferred} \\
\mathcal{U}((R_1 ; R_2), e) &:= \max(\mathcal{U}(R_1, e), \mathcal{U}(R_2, e)) && \text{both R_1 and R_2 must be satisfied} \\
\mathcal{U}(\{pred\}R, e) &:= \mathcal{U}(R, \inf\{t \geq e \ : \ pred(t)\})
\end{aligned} \tag{2.18}$$

The last formula simply says, that we pick the earliest time, when the *pred* is satisfied, to start evaluating the R. Each component count expression $H : P$, where $(H, P) \in \mathcal{B}(G)$, is computed simply as the number of all components $\boldsymbol{P}$ in the system at time t, or $\sum_{Q \in P \underset{*}{\bowtie} \_} v_{H,Q}(t)$.

The base case is n actions from the specified set of expected actions, $R^a[n]$. We are looking for the time, when the number of all these actions is higher by n in comparison with the start time e. Formally:

$$\mathcal{U}(R^a[n], e) := \inf\{t \geq e \ : \ \mathcal{U}'(R^a, e, t) \geq n\}$$

$$\text{for } \mathcal{U}'(a_1 \mid \ldots \mid a_k, e, t) := \textstyle\sum_{i=1}^{k} v_{a_i}(t) - \sum_{i=1}^{k} v_{a_i}(e).$$

2.8 Performance Trees

Performance Trees are a hierarchical tree structure representing performance queries in a simple graphical form. Their main purpose is to aid user's understanding of the query by the visual clarity. Apart from that, they are one of the most powerful formalisms developed, focused on testing both actions and states of the system behaviour. Their simple syntax makes them a feasible candidate for wider user adoption as well. A very informative summary of their options and examples of use can be found in [30], from where we borrowed our grammar and examples in this section.

One of the greatest advantages is, that they are independent of the model's formalism. They are abstract queries and not concerned with how the actual implementation will deal with them. That makes them easily portable between formalisms and simplifies their syntax even more, since they need no special constructs for handling models in any particular modelling formalism.

All of the operations used have already been developed or implemented in one way or another in other tools. That means they are perfectly feasible, which is proven by their existing full implementation (the Section 2.2). There appears to be no major problem with potentially incorporating them into other popular tools too.

As we said, Performance Trees are capable of very complicated performance queries and still are well comprehensible. A simple example can be:

What is the average time required to complete the passage defined by the convolution of the passage from the set of start states S1 to the set of target states T1 with the passage from the set of start states S2 to the set of target states T2, having the additional constraint that the set of states E is excluded from both passages?

When one reads this query, it is rather hard to comprehend and remember it, even though this is quite simple query. With Performance Trees, this is turned into a simple graph, as can be seen in Figure 2.6.

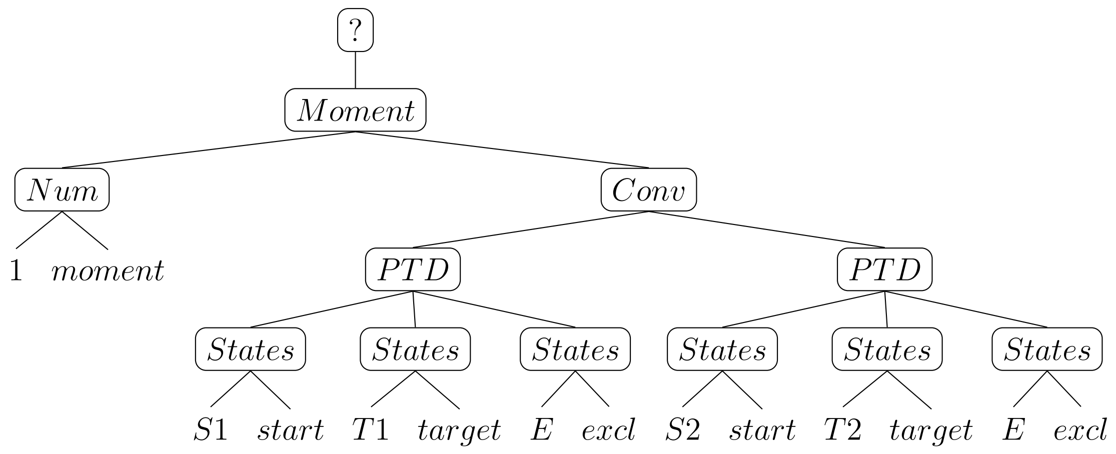

Figure 2.6: Motivation example

2.8.1 Syntax

Now we will introduce the syntax. The nodes in Performance Trees are of two different types - value nodes and operation nodes.

2.8.2 Value nodes

First, let us introduce the value nodes. These are the leaf nodes representing the concrete values, such as numbers, or sets of some objects.

We need a preliminary definition of two sets: $SAL = \{\text{state and action labels}\}$ and $TYP = \{start, target, incl., excl., time, prob., reward, moment, \emptyset\}$.

The **Bool** node represents either falsity or truth. We have that $Boolean \in \{true, false\}$.

$$Bool \quad \sim \quad Boolean$$

The **Num** node represents a numerical value. The second annotation, *Type*, represents the type of the numerical value. We define the types like this: $Integer \in \mathbb{Z}$, $Real \in \mathbb{R}$ and $Type \in TYP$.

$$Num \quad \sim \quad Integer,\ Type \ \mid \ Real,\ Type$$

The **Actions** node represents a set of actions. The first annotation contains the actions themselves, referenced by a set of labels, or directly. Labels identify actions by specifying conditions on the model. The second annotation describes the type of the action. We have $Action ::= a \mid tt \mid Action \wedge Action \mid \neg Action$, where $a \in SAL$ and $Type \in TYP$.

$$Actions \quad \sim \quad Action,\ Type$$

The **Sets** node represents a set of states. The first annotation contains the states themselves, referenced by labels, or directly. Labels identify states by specifying conditions on the model. The second annotation describes the type of the state. Here, $State ::= a \mid tt \mid State \wedge State \mid \neg State$,where $a \in SAL$, and $Type \in TYP$.

$$States \quad \sim \quad State,\ Type$$

Last value node represents a range (interval). It has two annotations, both of which are numerical.

$$\begin{aligned} [\![\ldots]\!] \quad ::= \quad & ([\ \oplus \mid Num \mid Moment \mid ProbInInterval \mid \\ & ProbInStates \mid SS{:}P \mid FR] \times (\oplus \mid Num)) \\ & \mid Moment,\ Moment \mid SS{:}P,\ SS{:}P \mid FR,\ FR \\ & \mid ProbInInterval, ProbInInterval \\ & \mid ProbInStates, ProbInStates \end{aligned}$$

2.8.3 Operation nodes

Operation nodes define mathematical or logical functions on their arguments. Inputs are their sub-nodes, which have a predetermined order. Outputs are then then plugged into their super-nodes, which are essentially operations on their results, with the exception of the root node, "?".

The "?" operator is the root node, which represents the overall result of the performance query.

$$\begin{aligned} ? \quad ::= \quad & ; \mid \vee\!\!\wedge \mid \neg \mid \bowtie \mid \oplus \mid PTD \mid Dist \mid Moment \\ & \mid Conv \mid InInterval \mid InStates \mid ProbInInterval \\ & \mid ProbInStates \mid SS{:}P \mid SS{:}S \mid FR \mid StatesAtTime \end{aligned}$$

The ";" operator is the sequential execution operator, which composes multiple performance requirements or metrics together into a single performance query. This operator is especially useful for the identification of optimisation opportunities across several sub-queries. It accepts at least two sub-nodes (sub-queries) as arguments. It returns a list of sub-queries combined.

$$\begin{aligned} ; \quad ::= \quad & (\ \veebar\!\!\wedge \mid \neg \mid \bowtie \mid \oplus \mid \mathit{PTD} \mid \mathit{Dist} \mid \mathit{Moment} \\ & \mid \mathit{Conv} \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{ProbInInterval} \\ & \mid \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{SS{:}S} \mid \mathit{FR} \\ & \mid \mathit{StatesAtTime}\ )^{2..*} \end{aligned}$$

The $\vee\!\wedge$ operator performs a boolean disjunction or conjunction operation. $\vee\!\wedge \in \{\vee,\ \wedge\}$. It has two truth value arguments.

$$\vee\!\wedge \quad ::= \quad (\vee\!\wedge \mid \neg \mid \bowtie \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{Bool})^2$$

The $\neg$ operator performs a boolean negation. It has one truth value argument.

$$\neg \quad ::= \quad \neg \mid \vee\!\wedge \mid \bowtie \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{Bool}$$

The $\bowtie$ operator performs a binary comparison. $\bowtie\ \in \{<,\ \leq,\ =,\ \geq,\ >\}$. Its arguments are numerical, but it returns a truth value.

$$\begin{aligned} \bowtie \quad ::= \quad & ([\ \oplus \mid \mathit{Num} \mid \mathit{Moment} \mid \mathit{ProbInInterval} \mid \\ & \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{FR}] \times (\oplus \mid \mathit{Num})) \\ & \mid \mathit{Moment, Moment} \mid \mathit{SS{:}P, SS{:}P} \mid \mathit{FR, FR} \\ & \mid \mathit{ProbInInterval, ProbInInterval} \\ & \mid \mathit{ProbInStates, ProbInStates} \end{aligned}$$

The $\oplus$ operator performs an arithmetic operation. $\oplus \in \{+,\ -,\ *, \div\}$. Its arguments and result are numerical.

$$\begin{aligned} \oplus \quad ::= \quad & ([\ \oplus \mid \mathit{Num} \mid \mathit{Moment} \mid \mathit{ProbInInterval} \\ & \mid \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{FR}] \times (\oplus \mid \mathit{Num})) \\ & \mid \mathit{Moment, Moment} \mid \mathit{SS{:}P, SS{:}P} \mid \mathit{FR, FR} \\ & \mid \mathit{ProbInInterval, ProbInInterval} \\ & \mid \mathit{ProbInStates, ProbInStates} \end{aligned}$$

The **PTD** operator represents a passage time density. Its arguments specify the passage and result is the density of time for a passage between two sets of states. Start and target sets of states are compulsory arguments, but optional constraints relating to actions (inclusion, exclusion), states (inclusion, exclusion) or rewards (represented by the range operator $[\![\ldots]\!]$) can also be supplied.

$$\begin{aligned} \mathit{PTD} \quad ::= \quad & \mathit{States}^{2..4} \mid \mathit{States}^{2..4},\ \mathit{Actions}^{1..2} \mid \mathit{States}^{2..4}, \\ & [\![\ldots]\!] \mid \mathit{States}^{2..4},\ \mathit{Actions}^{1..2},\ [\![\ldots]\!] \end{aligned}$$

The **Dist** operator represents a (cumulative) passage time distribution, which is obtained by converting a passage time density.

$$\textit{Dist} \quad ::= \quad \textit{PTD}$$

The **Conv** operator represents the convolution of two passage time densities or distributions.

$$\textit{Conv} \quad ::= \quad \textit{PTD, PTD} \;\mid\; \textit{Dist, Dist}$$

The **Moment** operator represents the raw moments of a passage time density function. It can calculate a moment of any order. The first argument is a number representing the order, second is **PTD**, **Dist** or a **Conv** that we calculate the moment for. This operator returns a numerical value.

$$\textit{Moment} \quad ::= \quad \textit{Num, PTD} \;\mid\; \textit{Num, Dist} \;\mid\; \textit{Num, Conv}$$

The **SS:P** operator represents the steady-state probability for a given set of states.

$$\textit{SS:P} \quad ::= \quad \textit{States}$$

The **SS:S** operator represents the set of states whose steady-state probability is equal to or in range of given probabilities. Its argument is a probability or an acceptable range of probabilities.

$$\textit{SS:S} \quad ::= \quad \textit{States, Num} \;\mid\; \textit{States}, \; [\![\ldots]\!]$$

The **FR** operator represents the average firing rate (and thus occurrences) of a certain transition.

$$\textit{FR} \quad ::= \quad \textit{Actions}$$

The **InInterval** operator determines whether a numerical value is within a certain interval or within multiple intervals and returns a truth value.

$$\begin{aligned}\textit{InInterval} \quad ::= \quad & [\, \textit{ProbInInterval} \mid \textit{Moment} \mid \textit{FR} \mid \oplus \\ & \mid \textit{SS:P} \mid \textit{ProbInStates}\,] \times [[\![\ldots]\!]]^{1..*}\end{aligned}$$

The **InStates** operator evaluates, if a certain state or set of states is contained in another set of states. Returns a truth value.

$$\textit{InStates} \quad ::= \quad \textit{States, States}$$

The **ProbInInterval** operator returns the transient probability with which the passage takes place in a certain amount of time, defined by a time range.

$$\textit{ProbInInterval} \quad ::= \quad \textit{PTD}, \; [\![\ldots]\!]^{1..*} \mid \textit{Conv}, \; [\![\ldots]\!]^{1..*}$$

The **ProbInStates** operator corresponds to the probability of being in a set of states at the time given, if we started in any of the states provided. The first argument is the set of start states, the second the set of target states and the third the time instant of interest.

$$\mathit{ProbInStates} \quad ::= \quad \mathit{States,\ States,\ Num}$$

The **StatesAtTime** operator returns the set of states that the system might occupy at a given time instant with a provided probability. The first argument is time instant, second the probability.

$$\mathit{StatesAtTime} \quad ::= \quad \mathit{Num,\ Num} \ \mid \ \mathit{Num,}\ [\![\ldots]\!]$$

This concludes the syntactic rules. For the type system, the following basic types are used:

num	:	a numerical value
range	:	a range of numerical values
bool	:	a truth value
func	:	a distribution or density function
states	:	states of the system
actions	:	actions that can occur in the system

The type system for the operators used in the formalism is is summarised as follows:

$$\begin{array}{rcl}
? & \vdash & (\mathit{bool} \mid \mathit{num} \mid \mathit{func} \mid \mathit{states})^{1..*} \\
; & \vdash & (\mathit{bool} \mid \mathit{num} \mid \mathit{func} \mid \mathit{states})^{2..*} \\
\bigvee\!\!\bigwedge & \vdash & \mathit{bool},\ \mathit{bool} : \mathit{bool} \\
\neg & \vdash & \mathit{bool} : \mathit{bool} \\
\bowtie & \vdash & \mathit{num},\ \mathit{num} : \mathit{bool} \\
\oplus & \vdash & \mathit{num},\ \mathit{num} : \mathit{num} \\
\mathit{PTD} & \vdash & \mathit{states}^{2..4} \mid \mathit{states}^{2..4},\ \mathit{actions}^{1..2} \mid \\
 & & \mathit{states}^{2..4},\ \mathit{range} \mid \mathit{states}^{2..4}, \\
 & & \mathit{actions}^{1..2},\ \mathit{range} : \mathit{func} \\
\mathit{Dist} & \vdash & \mathit{func} : \mathit{func} \\
\mathit{Conv} & \vdash & \mathit{func},\ \mathit{func} : \mathit{func} \\
\mathit{Moment} & \vdash & \mathit{num},\ \mathit{func} : \mathit{num} \\
\mathit{SS{:}P} & \vdash & \mathit{states} : \mathit{num} \\
\mathit{SS{:}S} & \vdash & \mathit{states},\ \mathit{num} \mid \mathit{states},\ \mathit{range} : \mathit{states} \\
\mathit{FR} & \vdash & \mathit{actions} : \mathit{num} \\
\mathit{InInterval} & \vdash & \mathit{num},\ \mathit{range}^{1..*} : \mathit{bool} \\
\mathit{InStates} & \vdash & \mathit{states},\ \mathit{states} : \mathit{bool} \\
\mathit{ProbInInterval} & \vdash & \mathit{func},\ \mathit{range}^{1..*} : \mathit{num} \\
\mathit{ProbInStates} & \vdash & \mathit{states},\ \mathit{states},\ \mathit{num} : \mathit{num} \\
\mathit{StatesAtTime} & \vdash & \mathit{num},\ \mathit{num} \mid \mathit{num},\ \mathit{range} : \mathit{states} \\
[\![\ldots]\!] & \vdash & \mathit{num},\ \mathit{num}
\end{array}$$

Chapter 3

Performance Trees using the Unified Stochastic Probes

In this chapter we will demonstrate how can we increase the expressivity of Performance Trees (the Section 2.8) and make them more intuitive by usage of the Unified Stochastic Probes formalism (the Section 2.5). In the Appendix A we also show that the renewed Performance Trees formalism is more powerful than the one proposed in the [30]. To prove that this system is feasible, it is enough to say, that our tool implements the Unified Stochastic Probes formalism thoroughly. The graphical notation is just a different representation of the same concepts. A tool that simply translates this representation to our defined Unified Stochastic Probes can easily be written and is planned as an extension of the GPAnalyser [29] tool.

3.1 Motivation

Previously proposed Performance Trees formalism offers a very expressive way of specifying our performance queries and requirements. It provides quantitative analyses over the obtained performance measurements specified in a simple and intuitive graphical manner, such as cumulative distribution of passage time density, their convolution or higher order moments. Behavioural (qualitative) performance requirements can be also easily specified, such as a passage between two sets of states.

In the previous versions however, it was very hard to specify individual passage time queries, both transient and steady-state ones. One needed to alter the model itself with an independent observer, called a probe, manually and monitor solely its actions using the PTD node. This process was very low-level and error prone. Furthermore, since we are interested in behaviour-oriented queries, such as a set of actions executed in a sequence, we are not usually interested in the states visited or reached. With the originally defined PTD node and also its extensions, it was necessary to specify not only the starting, but also the target states of the passage. If there are multiple possible target states after the given sequence of actions, which is very usual for a global passage in large-scale distributed systems, we need to specify all of them as unions of states. This brings even more complexity to specifying queries. Lastly, previously specified PTD node could only globally observe a set of actions required or forbidden to occur. This is very limited, as it does not allow to specify their ordering, or optional actions.

From this perspective, we can see the Unified Stochastic Probes with their expressiveness and simple syntax are far more powerful and easier to specify. That is the motivation for using them for these kinds of queries rather than the previously defined nodes.

3.2 Unified Stochastic Probes sub-trees

We are proposing a new way of specifying Unified Stochastic Probes as a sub-tree for PTD node instead of the previously defined arguments. We will simply add a new non-top level node called *Probe*. *PTD* will now take a *Probe* as its argument. Even though we could use *Probe* on its own, we also keep a *PTD* to allow alternative behavioural measures in the future. Below, we represent the overall *Probe* structure.

$$\mathit{Probe} \quad ::= \quad \mathit{GlobalProbe}, \mathit{LocalProbe}*, \mathit{Subst}*, (\mathit{Steady} \mid \mathit{Transient})? \tag{3.1}$$

The first annotation is the Global Probe, for which we propose the *GlobalProbe* node. It needs two parameters - both *PredActionSeq* nodes. First argument represents the behaviour to expect before starting the measuring (start signal is emitted, when executed), the second represents the stop signal. *PredActionSeq* closely follows Unified Stochastic Probes global probe grammar. The last optional argument is *Repeating*.

$$\begin{aligned}\mathit{PredActionSeq} \quad ::= \quad & \mid \; \mid \; ! \; \mid \; * \; \mid \; ? \; \mid \; + \; \mid \\ & , \; \mid \; ; \; \mid \; / \; \mid \; \varnothing \; \mid \\ & \epsilon \; \mid \; . \; \mid \; Iterate \; \mid \; Specific \; \mid \; \mathit{Predicate} \; \mid \; \mathit{Action}\end{aligned} \tag{3.2}$$

Probe operator nodes are either unary ('!', '$*$', '?', '+') represented as nodes with the respective symbol and one argument, or binary (',', '|', ';', '/' and '$\varnothing$') with two *PredActionSeq* arguments. *Iterator* operator is unary operator with extra one or two numerical arguments. Furthermore, we have a *Specific* node for specifying subsequent specific action (argument is the *Action*), ϵ node for empty sequence and '.' node for any action.

The *Predicate* nodes represent a logical predicate which has to be satisfied to advance the probe state. This closely follows the grammar of predicates as proposed in the Unified Stochastic Probes formalism (2.12), again separated into nodes. The second argument is a *PredActionSeq*. If there is a *Predicate* as the second argument, it represents a conjunction of predicates. $H : P$ represents plain group:component string denoting active instances of a component state P in group H. For simplicity, we also added more numerical and relational operations.

$$\begin{array}{lcl} \mathit{Predicate} & ::= & \mathit{pred}, \mathit{PredActionSeq} \\ \mathit{pred} & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg \mid \vee \mid \mathit{b\_expr} \\ \neg & ::= & \mathit{pred} \\ \vee & ::= & \mathit{pred}, \mathit{pred} \\ \mathit{b\_expr} & ::= & \mathit{r\_expr} \succeq \mathit{r\_expr} \\ \mathit{r\_expr} & ::= & \mathit{int} \mid H : P \mid \mathit{r\_expr} \oplus \mathit{r\_expr} \\ \succeq & ::= & < \mid \leq \mid > \mid \geq \mid = \\ \oplus & ::= & + \mid - \mid * \mid / \end{array} \tag{3.3}$$

Next arguments of the *Probe* can optionally be *LocalProbe* nodes, which represent local probes from the Unified Stochastic Probes formalism. Their first parameter is the name of the local probe. The second and further arguments are specified as *SignalAction* nodes. Their first argument is *ActionSeq* and the second argument is the signal emitted afterwards. Here, all the operator arguments are *ActionSeq* instead of *PredActionSeq*, since predicates are not allowed in local probes. Again, last optional argument is *Repeating*.

$$\begin{array}{lcl} \mathit{ActionSeq} & ::= & \mid \;\mid\; ! \;\mid\; * \;\mid\; ? \;\mid\; + \;\mid \\ & & , \;\mid\; ; \;\mid\; / \;\mid\; \varnothing \;\mid \\ & & \epsilon \;\mid\; . \;\mid\; \mathit{Iterate} \;\mid\; \mathit{Specific} \;\mid\; \mathit{Action} \end{array} \tag{3.4}$$

Next optional set of arguments for the *Probe* are nodes representing the substitutions, the *Subst* nodes. These have two arguments - the original component group of the system and the altered component group to replace it with (incorporating the previously defined local probes). The *Subst* nodes are applied in their specification order. The optional "[number]" after the component right in the Group represents replicated components working in parallel.

$$\begin{array}{lcll} \mathit{Component} & \sim & (\mathit{name} \mid \underset{\mathit{action\ names}}{\bowtie}), ([n])? & \\ \mathit{Subst} & ::= & \mathit{group}, \mathit{group} & \mathit{group} \text{ has a custom name} \\ \mathit{group} & ::= & \mathit{Component} & n \in \mathbb{Z}_+ \\ \underset{\mathit{action\ names}}{\bowtie} & ::= & \mathit{Component}, \mathit{Component} & \end{array} \tag{3.5}$$

The last *Probe* argument is the probe type node. This can be *Steady* for a steady-state individual passage probe, *Transient* for a transient individual passage probe or omitted, if we wish to determine the global passage. These need to follow the forms as specified in the Subsection 2.5.4.

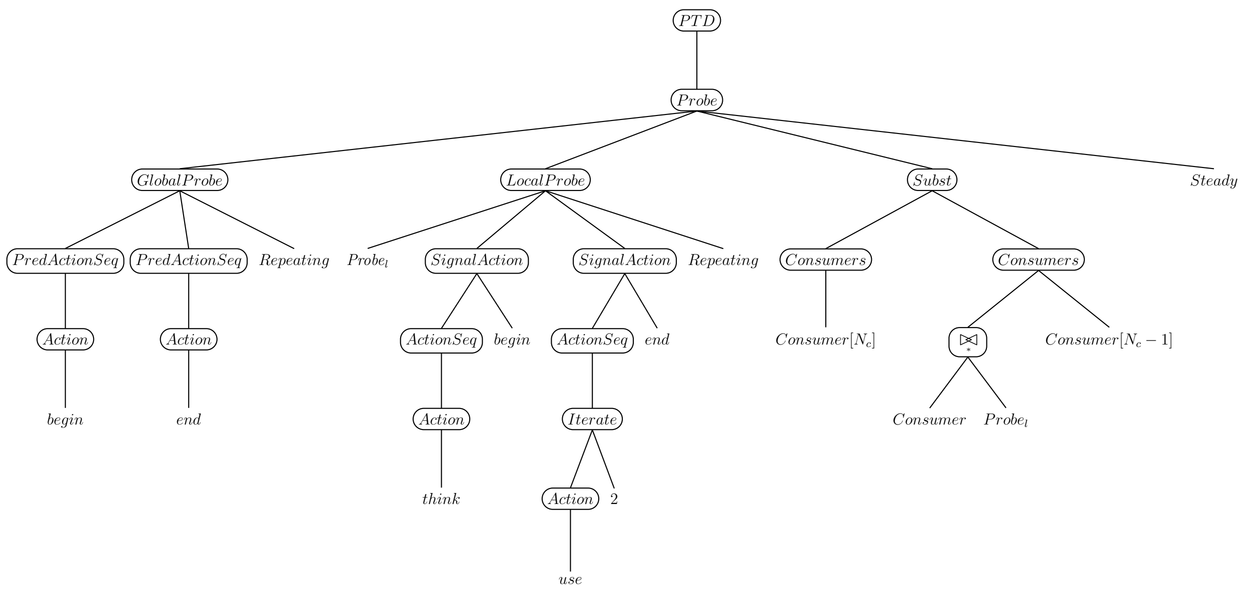

Figure 3.1: Steady-state individual passage-time probe example

3.3 Examples

In this section we will show concrete examples of how we can transform Unified Stochastic Probes to the new *Probe* node. We will use the examples in Subsection 2.5.5.

In the Figure 3.1 we demonstrate how the steady-state individual passage example can be translated with our proposed method. We start with an *PTD* node with a *Probe* child. Now we can split up the example probe definition into a global probe, local probe $Probe_l$, substitution and the "steady" flag. Each of these is a child of the *Probe* node. We follow by splitting the global probe start and stop expressions into the minimal subexpressions recursively and defining subtrees for them, which are then attached to our main tree. We do the same for the $Probe_l$ local probe. In this simple case, we have only one type of a local probe used, but this easily generalises to an unlimited number of local probe definitions. Substitutions are then also split into the components of the same type working in parallel (or possibly just one) and synchronizations are represented by internal tree nodes. Lastly, the "steady" flag needs no further arguments.

The transient individual passage example (Figure 3.2) can be mapped to a tree as follows. Again, we separate the definitions for the global probe, local probe $Probe_l$ and substitutions. In this case however, we have "transient" flag instead, again with no arguments. We recursively split the global and local probes definition, as well as substitutions in the same manner as in the previous example and attach them to the main tree.

The last example was for the global passage time. This closely follows the same procedure as the previous two examples, although there is no flag, since this is the default mode. Thus we have only a *PTD* node with a *Probe* node child. This in turn has only the global and the local probe, and the substitutions as its children.

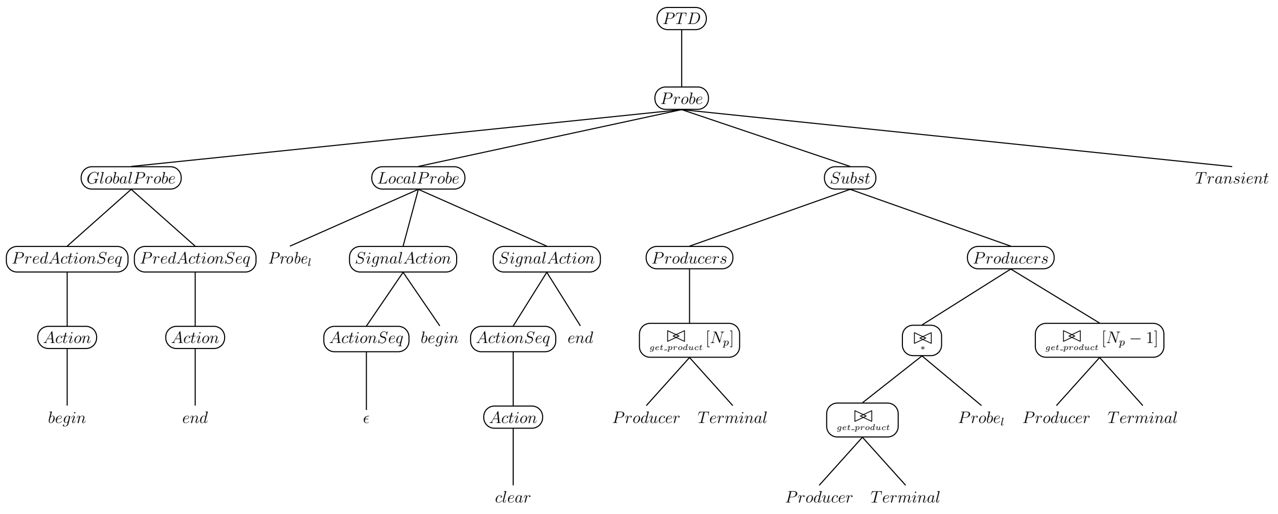

Figure 3.2: Transient individual passage-time probe example

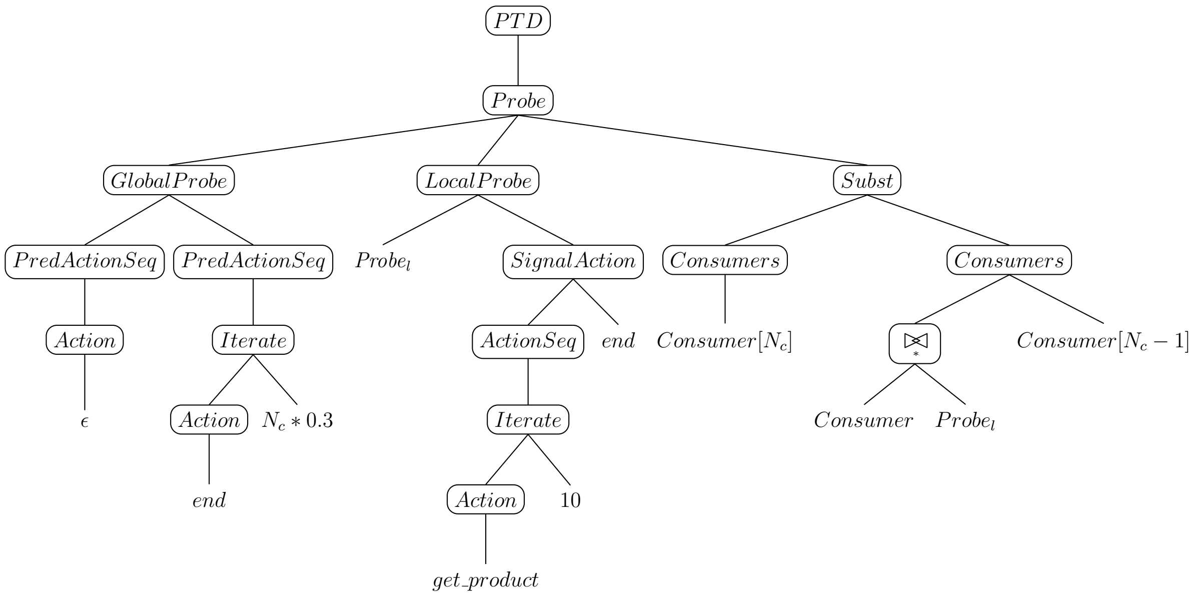

Figure 3.3: Global passage-time probe example

3.4 Alternative approaches

We have also considered alternative ways of specifying the PTD node using the Unified Stochastic Probes formalism. Namely, we could simplify the global and local probes sub-trees by not splitting their grammar into so many nodes. Rather we would just leave the original regex-based expression as their argument. That would make the trees much more concise and would possibly leave some more freedom to the user. But it also has disadvantages. Firstly, the explicit division into nodes allows better visualising the concepts and seeing the ordering and structure of the expression. Also, it orients tree downwards rather than to the sides, which makes it more comprehensible as a tree.

Another alternative approach was to split the regex syntax, as we did in the end, but leave the global probe predicates intact. This approach leaves the freedom to the user for predicates, but imposes the strict representation rules for the rest of the expression. That means the main expression maintains all the advantages of the strict syntax. On the other hand, the predicates are then very easy to specify with the "free" syntax (in the Probes predicates grammar boundaries). Although it might appear that this approach gives the best of the two worlds, we have to consider that this system is also more chaotic, since we do not adopt a unified approach. Also, predicates are used only for a particular kind of passage-time computation (currently only global passage time in the fluid-flow approximation mode) and therefore the freedom would be unlikely to have a massive impact on the usage of these Probe PTD nodes.

Lastly, we also considered letting the user to specify the substitutions in the original manner. Again, this would simplify the rules for the user for the price of less clear specification. To achieve the best user adaptability, we prefer the proposed approach.

Chapter 4

Implementation

In this section we present how we implemented a working parser for Unified Stochastic Probes as well as the passage time computations using both fluid-flow approximation and simulation. We summarise the technologies used, as well as the existing work - GPAnalyser capabilities. We also explain what we had to modify and introduce into this tool to prepare it for Probes computations. Furthermore, we talk about the newly added components and about increasing the current capabilities to accommodate for our needs and make much larger models tractable. Finally, we explain how to use our implementation in the Appendix B.

4.1 GPAnalyser

GPAnalyser tool ([29]) is a very powerful engine for performance evaluation of system models. It allows specifying models in restricted GPEPA. Originally, there was no support for passive actions, which we added along with the support for weighted passive actions (Section 4.7). GPAnalyser then translates this model to a population continuous-time Markov chain (PCTMC). Afterwards, it can analyse this PCTMC in both plain simulation and the faster fluid flow approximation mode. Using these techniques, it can estimate the mean component counts at all times, as well as their higher-order moments. Furthermore, it is capable of evaluating the moments for action occurrences and even user-specified rewards, such as engine fuel consumption over the time.

4.2 Architecture

In the Figure 4.1 we present the high-level view of the current version of GPAnalyser. The newly added components are highlighted in green, modified components in blue. Our main additions were the Unified Stochastic Probes parser and the Probes passage times computation engine, which is capable of working in both simulation and fluid flow approximation mode. We also present a novel method of overcoming Java method size limitation for projects, which need to generate large methods, by using C++. We implemented and used the C++ code output rather than existing Java and Matlab code outputs, which is used to either simulate the model (simulation mode) or approximate it (fluid flow approximation mode). This is done in an uninterrupted run of the tool and in no way interferes with inexperienced users.

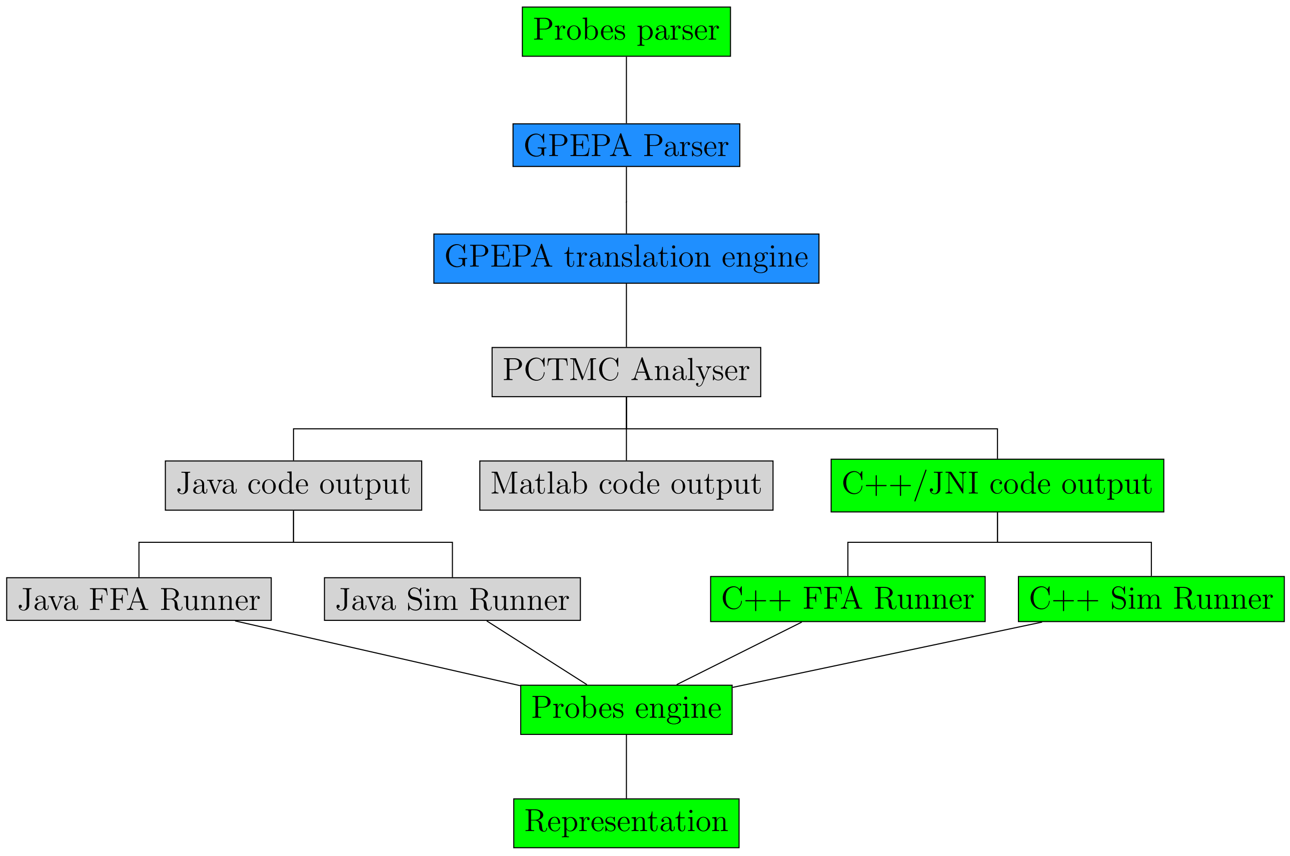

Figure 4.1: Simplified GPAnalyser architecture, the blue components were changed, the green added

Since we were extending the GPAnalyser tool, for simplicity of the project, we decided to follow up with the usage of most of the technologies rather than solve problems with interoperation between various execution platforms. That means we used Java for the main logic too. Small exception include Runge-Kutta ODEs (ordinary differential equation) solver for fluid flow approximation, which we rewrote to C++ to better utilise the performance, when using our novel C++ on the fly compilation. Our parser was also generated by ANTLR and extended the original GPEPA parser. Like the original authors, we adopted the unit testing JUnit framework.

4.3 Used libraries

The original GPAnalyser tool was using a lot of external libraries. Some of them were used directly in our extension. Furthermore, we have used some new libraries as well. Below we present a short introduction to these.

- *ANTLR* [2] is a compiler generator. One only specifies a grammar for a language in a simple manner and ANTLR transforms this into a powerful compiler in the target language, in our case Java. ANTLR can produce lexers for reading input tokens into streams and parsers for analysing the streams and, for simple languages, directly executing the desired behaviour. More preferably, since this is very limited, it can produce an Abstract Syntax Tree (AST). One can then generate a compiler, sometimes termed as the tree-walker, which parses the AST of the input and finally exhibits the required behaviour, or transforms the AST for another compiler. In our case, it simply walks the Unified Stochastic Probes ASTs and translates these into the iGPEPA language. This library is distributed under the BSD license.

- *Guava* [5] is a Java library, which supports various extensions to the original Java API. In our case, we only made use of the new collections, such as MultiSets, MultiMaps and BiMaps (namely their hash implementations). These were useful chiefly for simplifying the implementation of automata and their conversion. Guava is licensed with Apache License 2.0.

- *JFreeChart* [7] library is a free java library for producing various graphs and diagrams. The original project used it for all the graphical representations of the results and we continued the trend - our Probes output passage-time densities can (optionally) be seen in a JFreeChart graph. This library is distributed under the LGPL license.

- *Java Deep-Cloning library* [3] serves for deep cloning Java objects with all their member objects using the reliable Reflection technique. It was the perfect fit for copying large sets of automata, since we are using regex-like syntax and we have also some repetition operations, such as iterate. Again, this library is under the Apache License 2.0.

- *JUnit* [8] is a simple freely available unit testing framework for Java. It is integrated into many Java IDEs (Integrated Development Environment). It was used to automate the testing of probes parser and compiler. It is distributed under the Common Public License 1.0.

4.4 Technologies used

We have also used other technologies, which are required for running the project or which we used to improve the quality of the project during the development.

- *JDK* (Java Development Kit) [6] - GPAnalyser is relying on the Java technology introduced with Sun Java 6 (or JDK 1.6) called dynamic compilation, unlike most of the software in Java, which only requires JRE (Java Runtime Environment) to run. It essentially means that we can generate Java code from within a running Java program, compile it, load it and use it without restarting the program. We use this feature both for dynamic compiling of predicates (Subsection 4.6.3) in global probes and for generating and loading the native class wrappers for C++ (Section 4.8).
- *Apache Ant* [1] is used to provide a setup, building and running system for GPAnalyser. The original project relied on Ant rather than more Unix-oriented Makefile. It is easily integrable into Java IDEs and a well-portable. We extended the main file in order to fully support our extension.
- We also used *g++* (part of the [4]) for compilation of C++ code. As described in the Section 4.8, we did this to increase the tractable complexity of the models. It is an optional feature, which requires g++ to be set up on the host machine and accessible from the command line.

4.5 Parsing and compiling

In this project we have extended the original set of parsers and compilers for GPEPA language and also added the new Unified Stochastic Probes parser and compiler. This is demonstrated in the Figure 4.2.

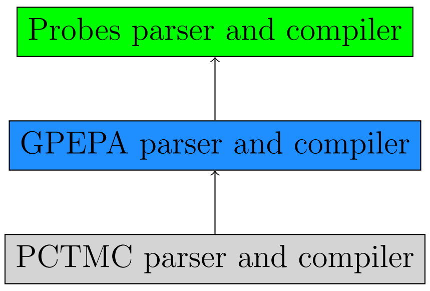
Probes parser and compiler
GPEPA parser and compiler
PCTMC parser and compiler

Figure 4.2: Simplified GPAnalyser parsers and compilators hierarchy, the blue components were changed, the green added

The parsers and compilers were all generated using the ANTLR library. We have chosen this library chiefly because the original GPAnalyser project used it and there has been a large reference code base reusable. In addition, all the parsers and compilers have their own, also gradually extended, lexer. With this architecture, we are able to separate concepts into different logical levels and substitute/add new parsers and compilers. For example, we could now add another stochastic process algebra language, or Stochastic π-calculus and easily adapt the Unified Stochastic Probes compiler for it simply by reusing this component.

Unfortunately, because of some ANTLR technical limitations, we were not able to absolutely separate Probes and GPEPA parser. ANTLR has difficulties with more than one level of grammar inclusion and it appears it is not yet well prepared for such complicated grammars. However, we did our best to keep Probes and GPEPA as separate as possible, although they reside in the same file, both parser and compiler.

We would like to better explain the workflow of ANTLR-generated compilers. As we have previously briefly mentioned, ANTLR can provide three basic grammar types - lexer, parser and compiler (or tree-walker) grammars. Although ANTLR grammar is limited to LL(*) grammars, it can optionally turn on back-tracking if necessary. This back-tracking is currently limited to one level, which is good for performance. An LL(*) grammar is the one which parses the input from left to right and uses the leftmost derivation of the input, with an unlimited number of look-ahead tokens.

- Lexer is the simplest grammar type. Its purpose is to filter out whitespaces and comments from the input and mainly, split the input into some reasonable tokens. These are then fed into a token stream.
- Parser serves for validating the token stream and for analysing the meaning of tokens sequences. It can execute some commands and one can use a parser directly to generate the full compiler for their language. Although this approach works, it does not utilise the full power of the ANTLR. Parser commands can only use the information which the parser has already processed. The more recommended approach is then to transform the simple token stream into an Abstract Syntax Tree (AST) with one's own defined rules. One can then write a tree walker manually, but much simpler approach is to use the third type of grammar in ANTLR. The advantage is, one can have multiple levels of tree-walking to do more complicated AST processing.
- Compiler, or tree-walker, is a grammar for parsing ASTs. It walks the AST from the root and executes our defined commands along the path. It uses depth-first walking of the AST. One can execute the commands when first encountering a node, between visiting its children or when leaving the node back to its parent. With the compiler grammar, one can write a full reliable compiler for a language adhering to the ANTLR limitations. One can also have multiple compilers, each parsing the AST, transforming it into a different AST and passing to the next compiler, although the “last” compiler needs to output the final code, for this process to be useful.

This workflow is demonstrated in the Figure 4.3.

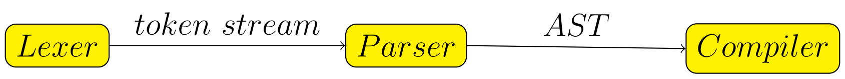

Figure 4.3: The basic ANTLR workflow

4.6 Parsing the Unified Stochastic Probes language

As we have stated, we used an ANTLR-generated compiler for parsing the Unified Stochastic Probes. We largely followed the rules of the [24], along with the Appendix C section. As the

Unified Stochastic Probes syntax is regex-based, this has not caused any major issues, since there has been a lot of research into this topic. Our main goal is to translate this regex syntax into the iGPEPA. Rather than doing that directly, which would diminish the portability of the Probes formalism to other process algebras; we first translate it into a deterministic finite state automaton (DFA). It is simple to map this onto an iGPEPA component, as demonstrated in Section 2.6. The focus of this section is therefore the translation of Probes syntax into a DFA. There are many existing algorithms for this process and we took our inspiration here.

However, rather than utilising existing libraries, we built a new set of algorithms for handling the automata, mainly because we had some operations, which are not standard. As we needed to build automata for these operations already, there was no major challenge in using these automata techniques for the standard operations as well, rather than adding a dependency on another external library and using its particular API.

For our finite-state machine, we adopted the OOP approach as compared to more widespread table-based approaches to set up paths between the states. A state is a basic unit and the simplest automata possible. In our package for automata, *uk.ac.imperial.doc.gpa.fsm* we represent it as a *NFAState* class. NFA emphasizes, that we are working with a non-deterministic finite automata. In our case, these surpass the deterministic finite automata (DFA), since, as discussed in Section 2.6, we ignore the failure state in DFAs.

States are then joint by various kinds of transitions. These are named after the actions from the probe definition directly. Normal transitions are standard *Transition* instances and will be later mapped to passive actions. Probe signals are mapped to *SignalTransition* instances. We treat these in a special way, since they will eventually be mapped into immediate iPEPA actions. Similarly, we have special types of transitions for ϵ, *EmptyTransition*, and '.' (an arbitrary transition), *AnyTransition*. The ϵ transitions will eventually be disposed of by converting the automata to a DFA. On the other hand, AnyTransition will be removed after generating the final DFA. In each state we will simply substitute it for all the actions our probe synchronises on with the observed component, except for those which have already been used in this state.

We also have two specialised SignalTransition classes, representing the start and stop signals in global probes respectively. Although, as discussed in [24], we do not wish to translate the global probes to iGPEPA for fluid-flow approximation, we do wish to do so for plain simulation of global passages. In the simulation mode, the cooperation over immediate actions between the global probe process and the model does not cause any trouble, even when they are each in their own group.

In the Figure 4.4 we present the complete workflow of our parser. Probe stands for any allowed Unified Stochastic Probe definition with the syntax as described in the user guide (Appendix B).

$$\begin{tikzcd}
\boxed{Probe} \arrow[r, "tokenise"] & \boxed{Tokens} \arrow[r, "parse"] & \boxed{AST} \arrow[r, "analyse"] & \boxed{DFA} \arrow[r, "translate"] & \boxed{iGPEPA}
\end{tikzcd}$$

Figure 4.4: Probes parsing and translation workflow

4.6.1 Translation of a probe expression

Probes use regex-based syntax and as such, they are naturally suited for recursion. Essentially, we can split these expressions into the minimal subexpressions, which are simply two states joint by one transition, as shown in the Figure **??**. Then gradually wrap them up in more complicated expressions for more complicated operations specified.

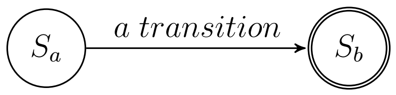

Figure 4.5: Actions (standard actions, signals, ϵ and .) mapped to a simple DFA

For each operation, except for the non-standard operations described in the Subsection 4.6.2, we then simply take its regex operand expression(s) and recursively analyse and convert them to NFAs. For the less complex operations, such as '$*$', we simply use this in a standard manner as described in the literature (e.g. for '$*$', we add an ϵ transition from all the accepting states to the initial state of the operand expression). For more complex operations, to save resources in the state space, we first use our conversion algorithm to convert the NFA operands to the minimal DFAs and then continue with the recursive approach. This way we can build the complete NFA when we reach the highest level. The only thing left to do is to convert this final NFA to a minimal DFA.

As discussed in the Section 2.6, we do not describe the conversion or minimisation algorithm here, as there are already many resources about this issue.

4.6.2 Non-standard operations

As explained in the Section 2.6, we do not describe the standard regex operations to automata conversion, as there are many materials about that already. However, we demonstrate how we tackled the non-standard operations, which are called “both” “reset” and “fail”. Again, “both” specifies, that both of the operand expressions have to be in an accepting state, in order for the overall expression to finish. “Reset” advances when the subexpression 1 gets to the accepting state, but restarts the evaluation to the initial state every time the subexpression 2 gets into an accepting state. “Fail” is the same as “reset”, except when the subexpression 2 reaches an accepting state, the whole expression fails and can never advance.

As described in [24, Appendix C], these operations can be handled by using the Cartesian product of the both expressions state spaces. Initial state is then defined as the state, where substate 1 and substate 2 are initial states of their respective subexpressions. Then we choose which states are accepting. For“ both” operation, it is every state when both its substates are accepting. For “fail” and “reset” it is all the states, where substate 1 is accepting and subbstate 2 is not. For each transition α from the alphabet, we choose the next state from (S_1, S_2) as (S_1', S_2'), where in the subexpression 1, α leads from S_1 to S_1' and in the subexpression 2 it leads from S_2 to S_2'.

For the “reset” operation, to handle the restart to the initial state, we simply add an ϵ transition from every (S_1, S_2), where S_2 is accepting, to the initial state. For the “fail” operation, we create a new failure state (with self-loops for every transition). Every (S_1, S_2), where S_2 is accepting, will then reach this state with an ϵ transition.

The next step is to convert this complex structure into a simple NFA. The simplest algorithm is to map each of the states (S_1, S_2), also the new failure state for the “fail” operation, to an unique NFAState and simply join them accordingly with the transitions from the Cartesian product.

These operations are handled directly in the Probes compiler with simple procedures for each of the operations. The common code is inside our helper *CartesianUtils* class from automata package. This class contains all the required operations for creating the Cartesian product state space from two NFAs, searching Cartesian product states and converting them to an NFA.

4.6.3 Global probes in the fluid flow approximation mode

As we have mentioned in the Subsection 2.7.3, the global probes supported operations in the fluid flow approximation mode have largely restricted grammar. In fact, the grammar is currently so limited, that we decided to specify special global probes grammar for this mode. It exactly follows the structure from the aforementioned subsection. ANTLR supports rule guards, which basically means we are able to switch between grammars conditionally. With this feature, for this particular mode of operation, we were able to impose restrictions of this grammar.

Another speciality is the predicates. They have their own subgrammar. We adopted the OOP approach here and each predicate is, after parsing, represented as a Java object. Since they have a lot of available operations and we would prefer them to be evaluable fast, as well as the simplest solution possible, we used the Java dynamic compilation for them. That means after they are parsed, the component group specifications are immediately substituted by the proper code for accessing them at various time instants. Then they are simply recompiled and loaded immediately. Although we could have also build a complicated set of classes with recursive functions for logical operators, this approach was much simpler and allows the predicates objects, represented by derivatives of the *NFAPredicate* class, to be easily copied or recreated. For the expressions, which have no predicates, we can use an *DummyPredicate* object, which always evaluates to true.

4.7 Modifications to the GPEPA translation engine

There were small changes to the GPEPA engine in order to fully support the Unified Stochastic Probes formalism. The foremost important change, applicable to also models in general, was introduction of the *passive* actions, which were originally lacking in GPAnalyser. These are absolutely essential for correctly using probes, since probes are independent observers and must not contribute to the rate of system actions in any way. Already in the process, we also implemented the *weighted passive* (Section 2.3) actions to allow for more interesting models specification.

The existing structure of the GPEPA engine allowed factorising the Prefix code to more classes. The original Prefix code, along with all the PEPA related code, used to be situated in the *uk.ac.imperial.doc.gpepa.representation.components* package. It was now split into *AbstractPrefix* with common code and interface, *Prefix*, which is a standard active prefix, *PassivePrefix* which is a weight-enabled passive prefix representation and also *ImmediatePrefix*.

ImmediatePrefix is very special. It is forbidden to be used directly. Rather it is used as a placeholder before the *vanishing state removal* (the Subsection 2.4.1) algorithm execution. This algorithm is included in the *PEPAComponentDefinitions* class. It has been implemented following the same steps, with one exception. Well-behaved components require only deterministic initial behaviour. For the purpose of implementing the probes, we did not require to support well-behaved components fully. Rather we assumed the model has deterministic all its signalling paths. This algorithm is called before the simulation or fluid flow approximation analyses are run on the model, in order to dispose of all the immediate actions in the model. They are still recorded as immediate paths executed right after an active or passive prefix, as described in the algorithm.

The main challenge was to implement all the possible interactions between the components. The basic structure had already been laid out for the cooperation of two active prefixes. We had to extend it to more cases, such as active and passive or weighted passive prefixes. This has been done for standard PEPA (*Cooperation* class). However, we also needed to implement the cooperation over immediate actions (hidden behind a non-immediate prefix). This needed to be done in order to support the nested local probes (so that they can observe the signal from another local probe). Likewise, we had to extend the GPEPA part of the code as well. Although, for fluid flow approximation, cooperation on signals leads to ill-formed system of ODEs equations (as discussed in [22]), this was essential for global passage times computation in the simulation mode, as in such case, the global probe is enclosed in its own group and cooperates with the main model in a different group, monitoring all the local probes.

4.7.1 Substitutions

Since local probes need to be attached to some component, we had to add another major algorithm, which searches the GPEPA model recursively, finds the group we are looking for and substitutes it for the group specified in the probe definition. Given the recursive nature of PEPA (and indeed GPEPA as well), the recursive searching is very simplified. What is more interesting about this algorithm is making sure we substituted the right component. Essentially, groups are uniquely defined in GPEPA by their label. Therefore we could, in theory, simplify the substitution process by just replacing the group specified with our new group using its label. No needs to specify the original group definition inside the probe definition or to pattern match the components to replace.

In our case, however, we decided to adopt the pattern matching approach. The reason is, that it ensures user does not make a mistake. Simply put, if the user specified the wrong group without pattern matching, there would be no means how to inform them, since our system would have no way of knowing. If we pattern match the group exactly though, and the group specification mismatches, because user made mistake in specifying the group label and we can immediately output an error.

4.8 Increasing the possible state space - C++/JNI dynamic compilation

The original GPAnalyser code handles the translation of GPEPA to a PCTMC. We heavily used this feature, as the PCTMC is the immediate format, with which GPAnalyser back-end works. After executing the conversion, GPAnalyser either works in the simulation mode or in the fluid flow approximation mode. The difference has been explained in the Chapter 2. What is important to note here is, that these two modes work in very similar way from the technical point of view.

In both of the modes, GPAnalyser generates an abstract code for transfer events. This code represents the model and it contains all the necessary mathematical operations to execute it and obtain the component counts, action counts and rewards of interest, at any given time. Naturally, this is different for fluid flow approximation and simulation. Later, GPAnalyser translates it into a concrete programming language. By default, this is Java but can be switched to Matlab via program arguments. In Java mode, the code is then directly compiled by GPAnalyser and used in the real-time (Java dynamic compilation, the Figure 4.6).

Java dynamic compilation is a feature of Java language and is naturally suited for similar applications, because it is faster than building complicated object-oriented model in memory. It allows using optimized Java bytecode instead. Unfortunately, there is a strong limitation in Java Virtual Machine official specification on the possible code size of any method. It basically forbids Java to compile any code, which contains a method larger than a certain amount of kilobytes (after translating to bytecode). Although this limit is set very high and people usually do not write so long code inside a single method (and it is by many software development practices regarded as a sign of a very poor design and/or code), it is often the case for automatically generated code to step over this limit. We run into the same problem with GPAnalyser. Essentially, for even averagely complex models, such as the working model from [24, Section V.], the generated rates expressions for events were extremely large and reflected the model intractable with the tool. This hindered our progress, as we were not able to evaluate the correctness of our algorithms.

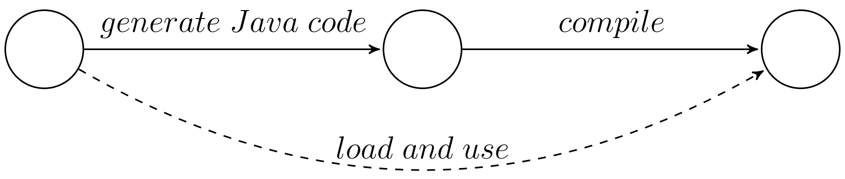

Figure 4.6: Java dynamic compilation workflow

One way how to solve this problem is to simplify the generated expressions beforehand. Although such an algorithm might nicely improve the performance during the execution of the model, it would be very long running initially. What is worse, model would potentially overgrow the limit anyway, unless separated into small methods very carefully. This would again introduce a lot of overhead. It is also a non-trivial task and would take a considerable time to implement.

We therefore adopted a different approach. Since C++ language has no such low method size limitation and was a good candidate also for fast execution, we decided to make use of it instead. Java supports a technique of calling natively written and compiled methods, which is called JNI (Java Native Interfaces). We could therefore write C++ output instead and compile it with g++. This is called automatically from GPAnalyser (and needs the host system to have g++ set up). Apart from the generated code, we also generate a JNI wrapper, which calls the native code. From inside GPAnalyser, we can therefore generate the C++ code for the model system, JNI wrapper for the native code and also Java class, which calls this JNI. Since Java supports dynamic class loading used for dynamic compilation, we can compile the generated Java class immediately, load it and used it in the same run. What we essentially obtained is a C++ dynamic compilation system used from Java. Informally, we call it C++/JNI dynamic compilation. This system is further demonstrated in the Figure 4.7. Since we alleviated the method size limit considerably, we are now able to work with much larger models.

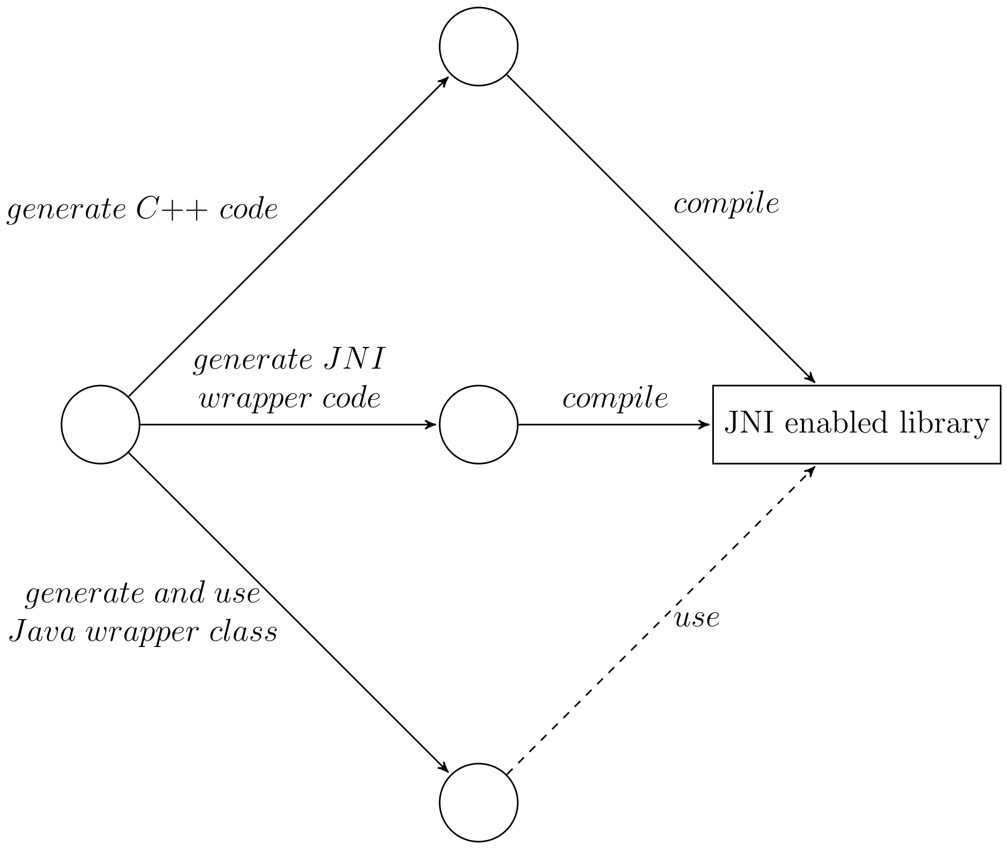
generate C++ code
compile
generate JNI wrapper code
compile
JNI enabled library
generate and use Java wrapper class
use

Figure 4.7: C++/JNI compilation workflow

We wrote all the required code for both fluid flow approximation and simulation. They both use the same C++ code printer for translating the abstract code obtained from GPAnalyser. They differ in the execution itself however, in the same way as the original Java dynamic compilation mode differed for these two modes. The runners are also implemented. They have virtually the same interface as the Java runners (simulation or fluid flow approximation respectively) and therefore can be easily switched over. We therefore made the C++ compilation an optional feature switchable from command line.

We have to point out that there are still limitations with this approach. GPAnalyser first generates the target language code in the RAM memory and then outputs it into a file. If the code is too large, it might outgrow the Java memory heap and essentially crash the JVM. However, on the modern computers with lots of RAM memory, this is considered unlikely.

4.9 Probes engine

We have also written the engine to execute the Probes and evaluate all the passage time densities described in the Subsection 2.5.4, namely steady-state individual, transient individual and global passage time density. We are directly using the GPAnalyser capabilities to evaluate component and action counts at certain times, for both the simulation and fluid flow approximation. We can use either our C++/JNI setup, or equivalently the original Java setup. In our package *uk.ac.imperial.doc.gpa.probes* we have an abstract class *AbstractProbeRunner* to specify the interface for running our probes. It also unifies the code for running both simulation and fluid flow approximation of GPAnalyser. This is then extended by *SimProbeRunner*, which runs the probes using the GPAnalyser runner set to simulation mode and then evaluates the passage times according to the Subsection 2.5.4. *ODEsProbeRunner* works similarly, although uses GPAnalyser in fluid flow approximation mode and uses the algorithms from the Subsection 2.7.3.

For the steady-state individual passage density in simulation, we therefore set the model up with the substitutions as specified in the probe definition, which means we attach a repeating local probe to the component of interest and run the simulation for a very long time. Then we use an equivalent probe definition with a non-repeating local probe and attach it to the component in the state we left it in in the last simulation. We generated this definition while compiling the probe with our compiler, since it is a simple change of one flag (repeating). We also remember the original GPEPA model so that we can substitute the non-repeating (absorbing) probe as well. When we enter the Probe engine, we simply remember two models with their respective substitutions applied. We keep the component counts for all the unprobed components and use the probabilistic approach for the probed component assignment. Afterwards, we just rerun the simulation and obtain the time, when our local probe is in an accepting state. Repeating the experiment then provides the density. For the fluid flow approximation, we keep the two models generated. After the first run, we use the obtained data to initialise the second model with the method described in Subsection 2.7.3, run it and sum the probabilities of being in a local probe accepting state at various times, as described in the aforementioned section.

Transient individual passage time in simulation mode is even simpler. We attach the probe to our individual, run the simulation and just wait till the local probe is in an accepting state. Naturally, we ignore the time until the `begin` signal has been emitted. Again, multiple runs of the experiment then provide the density function. In fluid flow approximation, we in theory should run the model until the infinite time initially and then for each of the time instants, run the model from that instant, as described in the Subsection 2.7.3. However, as discussed in that section, we do not need to do so, since with the time approaching the infinity; the integral inner term weighting is decreased rapidly and converging to zero. We can therefore obtain a fixed point of the weighting and run only up to that time instant. Afterwards, we just recover the unconditional CDF. Unfortunately, current version of GPAnalyser does not support fixed point solving and we have to estimate manually the good time for truncation of the integral. For the user, it might be simpler to just set very a long time and wait, since the fixed point/manual time truncation is only about reducing the waiting time.

Lastly, for global passage, we first attach the local probes as specified in substitutions (if any). Then we translate the global probe to the NFA and later minimal DFA, just as with local probes. Afterwards, we simply translate it to the iPEPA and wrap in its own group. We also wrap the main model, make them synchronized and run the simulation. When the `stop` signal is sent, we record the time and rerun the simulation with the same set up. This way we efficiently obtain the density. For this mode, we ignore the predicates, as discussed in the Subsection 2.5.4.

For fluid flow approximation we adopt a completely different approach. We wrote a set of classes representing the recursive $\mathcal{U}(R, e)$ function from the Subsection 2.7.3, which is situated in the *uk.ac.imperial.doc.gpa.probes.GlobalProbesExpressions* package. There are classes representing each of the operations (“both”, “choice”, “sequence” and predicated expression). Also we have a class to represent $\mathcal{U}'(R^a, e, t)$ expression and simple "action". We evaluate them using the visitor pattern, which is naturally suited for evaluation of recursive expressions. This allows simple potential extensions to other operations in the future. When using our modified ANTLR grammar specific for global passage time in this mode, rather than building the global probe automata, we directly build up the overall expression (where the highest level is the sequence between the expression preceding the start signal and the one preceding the stop signal). Then we simply run the fluid flow approximation on the main model with applied substitutions as specified in the probe definition and evaluate this expression on the resulting snapshots of the system.

4.10 Representation

After we run a probe, we would like to show the user a nicely formatted output. We have written a simple class *CDF* in our *uk.ac.imperial.doc.gpa.probes* package, which represents the final CDF obtained from passage time computations. As we have unified their output into a simple class representation, we can now write many different convertors of this class into any target output.

The basic output is a graph representation, which we generate using the JFreeChart library. It is a simple graph mapping time to probability. Conciseness was the main idea, as the user is most interested in this metric. This feature is optional.

For more advanced users we also prepared a simple textual output. With a special syntax construct inside the probe definition, they can easily specify the output to be a file on their hard-drive. This is not mutually exclusive with the graphical output and the user can choose to have both outputs. The textual output are simple time-probability pairs per each line and therefore easy to convert to any required format with a simple script. In addition, many graphic plotters, such as gnuplot or pgfplot (a Latex library) can work directly with this format.

4.11 Overall workflow

We would like to illustrate the overall workflow of working with probes. This is demonstrated in the Figure 4.8. We provide GPAnalyser with a model file. This includes the iGPEPA model of interest, possibly some other metrics or rewards, and finally, a probe or even multiple probes definitions. GPAnalyser parses this file and builds up an overall iGPEPA model, with substitutions from the probe applied. Then we remove the vanishing states and obtain a more basic GPEPA model. This is translated to a PCTMC, which GPAnalyser uses as the intermediate form portable between the most process algebra formalisms. Then, depending on specifying the simulation or fluid flow approximation, the appropriate mathematical model is generated in an abstract syntax. As discussed in the Section 4.8, this is then translated to some concrete programming language. In case of Java or C++, this is taken even further. They are dynamically compiled and the model is solved. Afterwards, using the results, we analyse the passage time and present the results in the form of CDF.

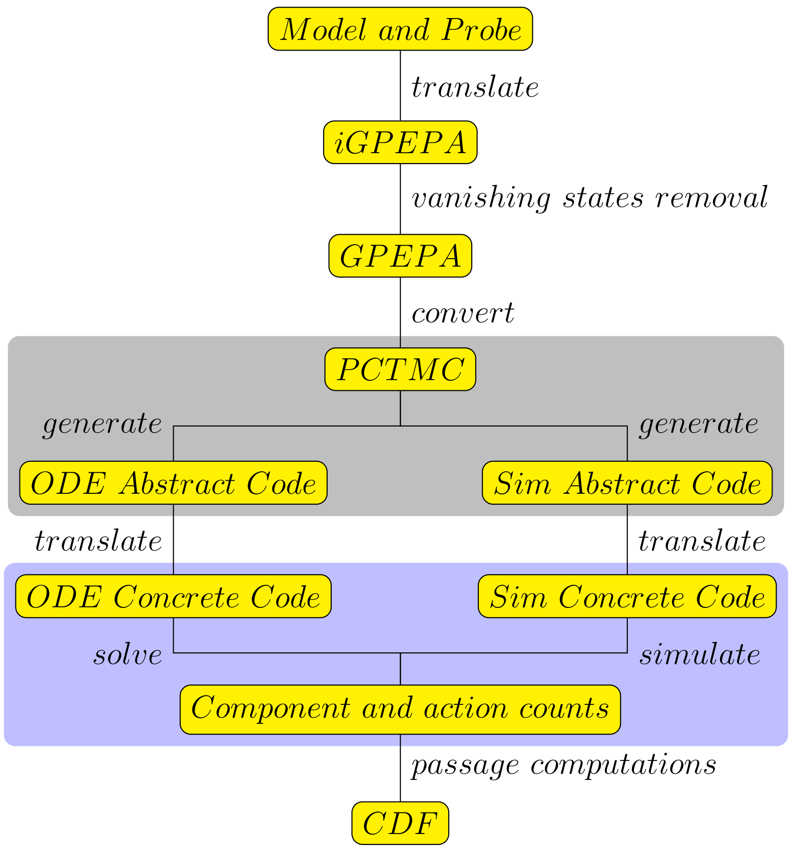

Figure 4.8: Complete workflow of a probe. Grey parts were provided by original GPAnalyser. Blue parts were provided in their Java version and we added the C++ version.

Chapter 5

Evaluation and future work

In this chapter we discuss how we validated the project, both data and the code. First we elaborate on the techniques used to ensure the code correctness. Then we prove the data validity by comparing our results with those hand-crafted from the [24]. Lastly, we demonstrate the power of this implementation by analysing some real-life models.

5.1 Code validation

In this project we used multiple ways to validate the code. The original GPAnalyser [29] project heavily used unit testing and we also adopted this practice for our extension. Since writing unit tests with a complete coverage of all the new code is an enormous effort demanding in time, we took another approach.

The unit tests are thus used only to verify the new probes parser. The technique is simple - we provide a text file with a local probe definition. This is then parsed with our parser up to the point, when we create a PEPA representation. This representation is then compared with the reference PEPA model prepared for this probe. We have multiple test probes, each targeting some probe operation. This way, we do test all the algorithms in the probe translation process at once. The disadvantage is, that if there were an error in the translation, we would have to delve deeper into the process to find the exact cause. However, due to the code structuring into small blocks, this has not caused a major trouble.

The rest of the added and modified code, especially dynamic C++/JNI compilation, could also be unit tested. Although it would be possible, it would require significant time. Also, these algorithms were often continually improved and changed to extend the expressiveness power or provide better performance. It was decided writing unit tests during this process is not a wise time investment, since the unit tests would have to be changed too often. Therefore, for the overall results we used the acceptance testing. For that, we directly compared our data to the reference hand-crafted data from [24].

5.2 Data validation

Passage time computations over PEPA models generate a lot of analysable data. The most interesting ones for us were the passage time density itself. We implemented both textual and graphical representation output. Textual representation could be compared to the reference data in an automatic manner. The original models and their passage time calculations from [24] were hand-craftily implemented in Matlab, both the simulation and fluid-flow approximation using ODEs. In this section, we show our automated implementation in the direct comparison with these manually generated results, which we obtained with the original authors' permission.

5.2.1 Comparison of the simple Client/Server model

First we will show the simpler Client/Server model and a sample Unified Stochastic Probe operating on it from [24].

The model was defined as:

$$\begin{aligned}
\boldsymbol{Client}_0 &\overset{def}{=} (fetch, r_t).\boldsymbol{Client}_1 & \boldsymbol{Client}_1 &\overset{def}{=} (reset, r_s).\boldsymbol{Client}_0\\
\boldsymbol{Serv}_0 &\overset{def}{=} (initialise, r_i).\boldsymbol{Serv}_1 & \boldsymbol{Serv}_1 &\overset{def}{=} (fetch, r_t).\boldsymbol{Serv}_0\\
\boldsymbol{Serv}_2 &\overset{def}{=} (recover, r_r).\boldsymbol{Serv}_0 & &\quad + (fail, r_f).\boldsymbol{Serv}_2
\end{aligned}$$

$$\begin{aligned}
&\boldsymbol{SC}(n, m) \overset{def}{=}\\
&\underbrace{(\boldsymbol{Client}_0 \parallel \ldots \parallel \boldsymbol{Client}_0)}_{n} \underset{\{fetch\}}{\bowtie} \underbrace{(\boldsymbol{Serv}_0 \parallel \ldots \parallel \boldsymbol{Serv}_0)}_{m}
\end{aligned}$$

The first sample probe was a steady-state individual passage probe to compute a time density till a recovered server will fetch twice without failing.

$$\begin{aligned}
\boldsymbol{PM}_6 &\overset{def}{=} \mathsf{begin : start, end : stop}^{\hookleftarrow}\\
&\mathsf{observes}\\
&\qquad \boldsymbol{Probe}_l \overset{def}{=} recover : \mathsf{begin}, (fetch[2])\backslash fail : \mathsf{end}^{\hookleftarrow}\\
&\mathsf{where} \quad \mathbf{Servers}\{\boldsymbol{Serv}_0[?n]\} \Longrightarrow\\
&\qquad \mathbf{Servers}\{(\boldsymbol{Serv}_0 \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{Serv}_0[?n-1]\}\\
&\mathsf{in} \quad \boldsymbol{SC}(n, m)
\end{aligned} \tag{5.1}$$

Here we present the original results compared with our implementation. The parameters for the model were $r_t = 0.4$, $r_s = 0.15$, $r_i = 0.6$, $r_r = 0.35$ and $r_f = 0.1$. The numbers of generated ODEs in our implementation were 13 and 16 respectively. The actual computation of the fluid-flow approximation took us about 10 seconds, whereas simulation with 5000 repetitions, minutes, depending on the model size.

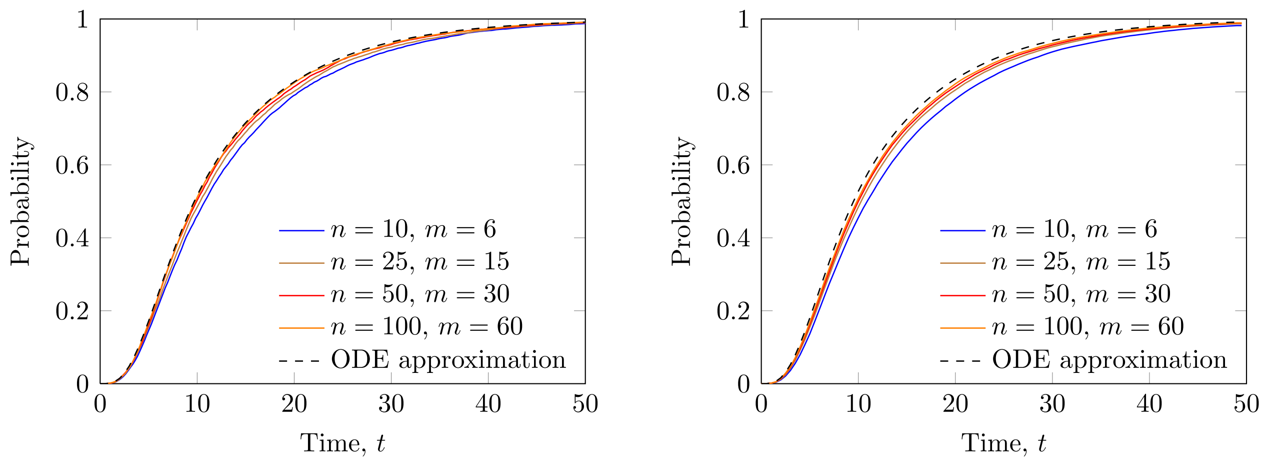

Figure 5.1: Reference and our implementation results for simple Client/Server model

We also present a direct comparison. One can easily notice that although the results are similar, there are some marginal differences. After consultations with the original authors we determined, that the difference is caused by the different method they used to quicken the computation of this example. They first ran the unprobed version of the model for certain time (until equilibrium). Then they remembered the component counts and added one probed server in its state after `end` has been fired, effectively executing 61 servers rather than 60 for the second step of the algorithm. The ratio 1 : 60 was favourable enough to introduce only small approximation error.

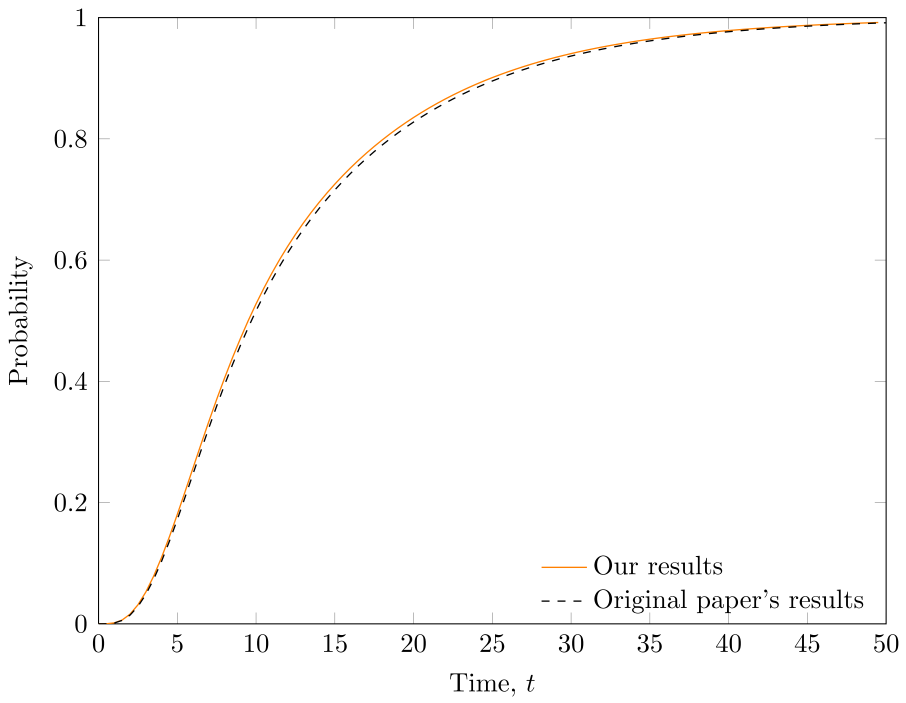

Figure 5.2: Reference and our implementation results for simple Client/Server model

5.2.2 Comparison of the worked example

Now we can demonstrate the main worked example from the [24]. This was a more complex model representing a large wireless sensor network of autonomous battery-powered agents. The GPEPA model was specified as follows.

The main agent component was a simple component with multiple possible actions.

$$\begin{aligned}
\boldsymbol{ClientHibernate} &\overset{def}{=} (clt\_on, r_{on}).\boldsymbol{ClientStandby} \\
&\quad + (clt\_shutdown, \top).\boldsymbol{ClientHibernate} \\
\boldsymbol{ClientStandby} &\overset{def}{=} (clt\_off, r_{off}).\boldsymbol{ClientHibernate} \\
&\quad + (radio\_init, r_{init}).\boldsymbol{ClientRadioUse} \\
&\quad + (cont\_tfr, r_{cont}).\boldsymbol{ClientStandby} \\
&\quad + (clt\_shutdown, \top).\boldsymbol{ClientHibernate} \\
\boldsymbol{ClientRadioUse} &\overset{def}{=} (data\_tfr, r_{radio}).\boldsymbol{ClientStandby} \\
&\quad + (clt\_shutdown, \top).\boldsymbol{ClientHibernate}
\end{aligned}$$

Various actions are possible with various levels of battery.

$$\begin{aligned}
\boldsymbol{BatteryEmpty} &\overset{def}{=} (clt\_charge, r_{charge}).\boldsymbol{Battery}_1 \\
\boldsymbol{Battery}_0 &\overset{def}{=} (clt\_shutdown, r_{shutdown}).\boldsymbol{BatteryEmpty} \\
\boldsymbol{Battery}_i &\overset{def}{=} (data\_tfr, \omega_{tfr} \times \top).\boldsymbol{Battery}_{i-1} \\
&\quad + (data\_tfr, \top).\boldsymbol{Battery}_i \\
&\quad + (cont\_tfr, \omega_{tfr} \times \top).\boldsymbol{Battery}_{i-1} \\
&\quad + (cont\_tfr, \top).\boldsymbol{Battery}_i \\
&\quad + (radio\_init, \omega_{init} \times \top).\boldsymbol{Battery}_{i-1} \\
&\quad + (radio\_init, \top).\boldsymbol{Battery}_i \\
&\quad + (clt\_off, \top).\boldsymbol{Battery}_i \\
&\quad + (clt\_on, \top).\boldsymbol{Battery}_i \\
&\quad + (clt\_charge, r_{charge}).\boldsymbol{Battery}_{\min(N_b, i+1)} \\
&\qquad\qquad : \text{for } 1 \leq i \leq N_b
\end{aligned}$$

To each agent, we then connect a single battery parameterised with N_b, which represents the maximum capacity of the battery.

$$\boldsymbol{CB} \overset{def}{=} \boldsymbol{ClientHibernate} \underset{L_1 \cup L_2}{\bowtie} \boldsymbol{Battery}_{N_b}$$

Lastly, we need a communication channels for the agents:

$$\begin{aligned}
\boldsymbol{Chan} &\overset{\mathit{def}}{=} (\mathit{data\_tfr}, r_{\mathit{radio}}).\boldsymbol{ChanBusy}_1 \\
&\quad + (\mathit{cont\_tfr}, r_{\mathit{cont}}).\boldsymbol{ChanBusy}_2 \\
\boldsymbol{ChanBusy}_1 &\overset{\mathit{def}}{=} (\mathit{data\_tfr}, r_{\mathit{radio}}).\boldsymbol{Chan} + (\mathit{tmt}, r_{\mathit{tmt}}).\boldsymbol{Chan} \\
\boldsymbol{ChanBusy}_2 &\overset{\mathit{def}}{=} (\mathit{cont\_tfr}, r_{\mathit{cont}}).\boldsymbol{Chan} + (\mathit{tmt}, r_{\mathit{tmt}}).\boldsymbol{Chan}
\end{aligned}$$

And the overall system is then defined as cooperation of the clients using the provided channels:

$$\boldsymbol{DWN}(N_c, N_h) \overset{\mathit{def}}{=} \mathbf{Clients}\{\boldsymbol{CB}[N_c]\} \underset{M}{\bowtie} \mathbf{Net}\{\boldsymbol{Chan}[N_h]\}$$

Finally, we can define a sample steady-state individual passage probe, which determines how long till an agent shuts down due to the battery, given that it already has sent some control information and data without shutting down.

$$\begin{aligned}
&\boldsymbol{SC} \overset{\mathit{def}}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\quad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{\mathit{def}}{=} \\
&\qquad (\mathit{data\_tfr}; \mathit{cont\_tfr}) \backslash \mathit{clt\_shutdown} : \mathsf{begin}, \\
&\qquad \mathit{clt\_shutdown} : \mathsf{end}^{\hookleftarrow} \\
&\quad \mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{CB}[?n]\} \Longrightarrow \\
&\qquad \mathbf{Clients}\{(\boldsymbol{CB} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{CB}[?n-1]\} \\
&\quad \mathsf{in} \quad \boldsymbol{DWN}(N_c, N_h)
\end{aligned}$$

Here we present the original results compared with our implementation. The parameters for the model were $r_{on} = 0.3$, $r_{off} = 0.6$, $r_{init} = 0.5$, $r_{cont} = 0.85$, $r_{radio} = 0.16$, $r_{charge} = 0.02$, $r_{shutdown} = 1.5$ and $r_{tmt} = 0.2$; weights: $\omega_{init} = 0.07$ and $\omega_{tfr} = 0.15$; and $N_b = 3$. The numbers of generated ODEs in our implementation were 59 and 71 respectively. The actual computation of the fluid-flow approximation took us about one and a half minute. Simulation with, 50000 repetitions, minutes or tens of minutes up to few hours.

Another example for a probe is a transient individual passage probe measuring sending four data packages from the time a probe sends a control package.

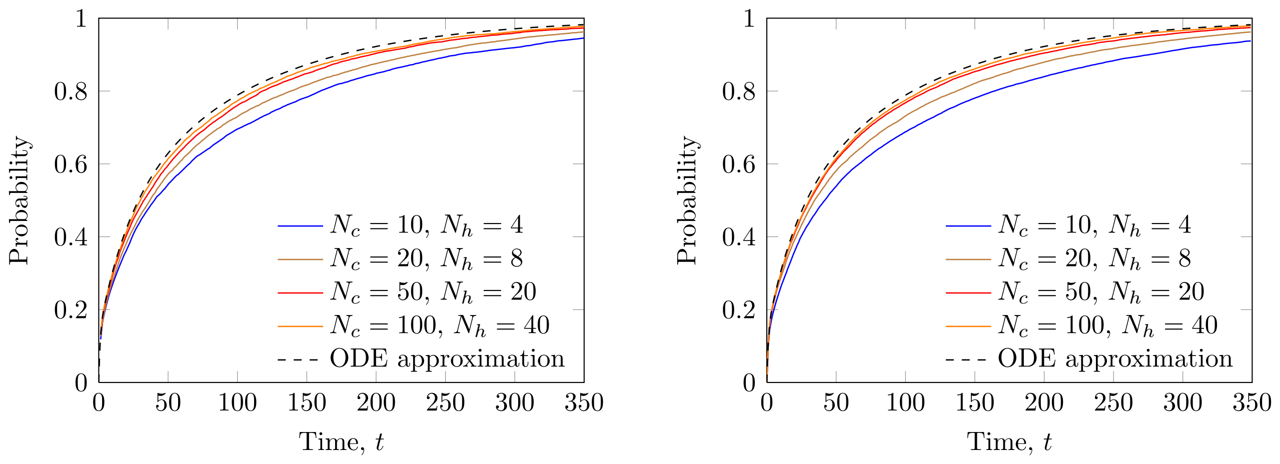

Figure 5.3: Reference and our implementation results for worked example model - steady-state individual passage time

$$\boldsymbol{SC} \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}$$

$$\mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=}$$

$$\mathit{cont\_tfr} : \mathsf{begin}, \mathit{data\_tfr}[4] : \mathsf{end}$$

$$\mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{CB}[?n]\} \Longrightarrow$$

$$\mathbf{Clients}\{(\boldsymbol{CB} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{CB}[?n-1]\}$$

$$\mathsf{in} \quad \boldsymbol{DWN}(N_c, N_h)$$

Below we show the original results compared with our implementation. The parameters for the model were $r_{on} = 0.3$, $r_{on} = 0.3$, $r_{off} = 0.6$, $r_{init} = 0.5$, $r_{cont} = 0.85$, $r_{radio} = 0.35$, $r_{charge} = 0.05$, $r_{shutdown} = 1.5$ and $r_{tmt} = 0.2$; and weights: $\omega_{init} = 0.07$ and $\omega_{tfr} = 0.15$; and $N_b = 3$. The number of generated ODEs in our implementation was 87. The actual computation of the fluid-flow approximation took us about one minute. Simulation with, 50000 repetitions, minutes or tens of minutes up to few hours.

The last example in the paper was a global probe measuring a time until a half of the agents shuts down at least once.

$$\boldsymbol{SC} \overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_c/2] : \mathsf{stop}$$

$$\mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} \mathit{clt\_shutdown} : \mathsf{end}$$

$$\mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{CB}[?n]\} \Longrightarrow$$

$$\mathbf{Clients}\{(\boldsymbol{CB} \underset{*}{\bowtie} \boldsymbol{Probe}_l)[?n]\}$$

$$\mathsf{in} \quad \boldsymbol{DWN}(N_c, N_h)$$

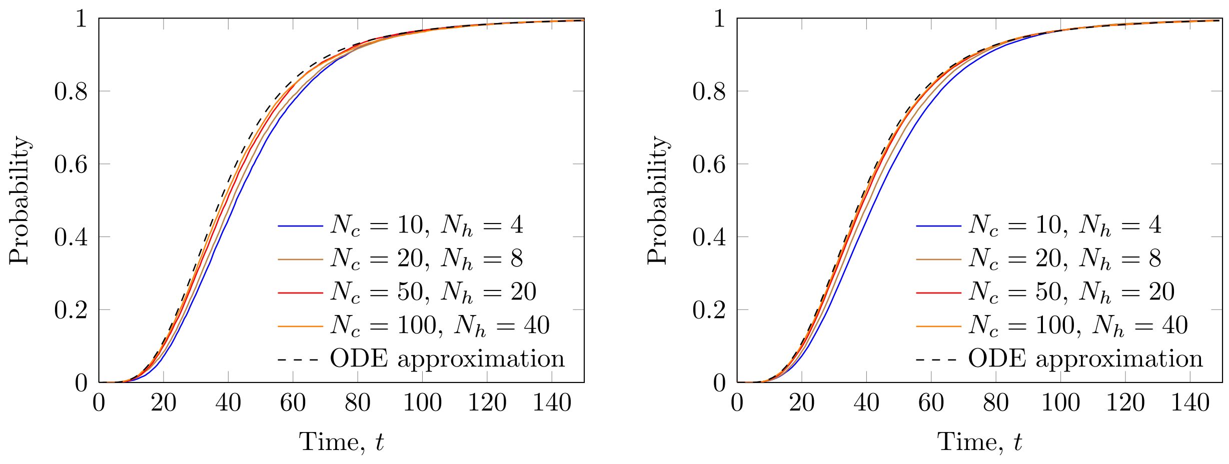

Figure 5.4: Reference and our implementation results for worked example model - transient individual passage time

The comparison of the results is presented below. The parameters for the model were $r_{on} = 0.3$, $r_{off} = 0.6$, $r_{init} = 0.5$, $r_{cont} = 0.85$, $r_{radio} = 0.4$, $r_{charge} = 0.1$, $r_{shutdown} = 1.5$ and $r_{tmt} = 0.2$; and weights: $\omega_{init} = 0.07$ and $\omega_{tfr} = 0.15$; and $N_b = 3$. The number of generated ODEs in our implementation was 28. The actual computation of the fluid-flow approximation took us less than 10 seconds. Simulation with, 5000 repetitions, minutes or tens of minutes, depending on the model size.

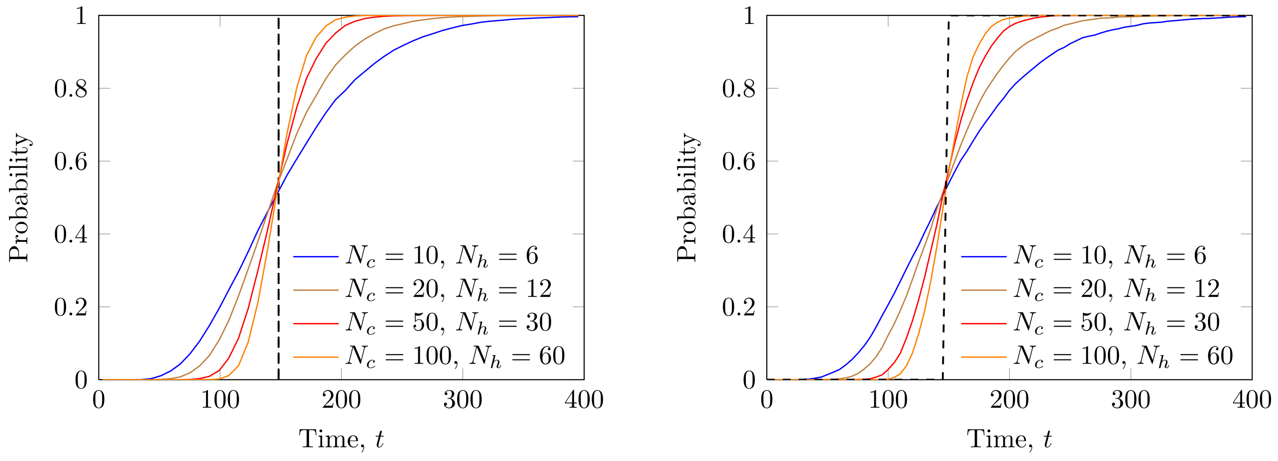

Figure 5.5: Reference and our implementation results for worked example model - global passage time

5.3 Real-life models

In this section we describe two models we devised for testing and evaluation purposes. The first is a very simple model, but it fully demonstrates the new iGPEPA capabilities, namely passive, weighted passive and immediate transitions. The second model is more complex and has much greater state space, especially after applying the probes. Without our C++/JNI dynamic compilation, we would not be able to analyse this model.

5.3.1 An example web-based application

For the purpose of demonstration, we modelled a client-server application. The clients request certain complex computations. They either get hold of a server, in which case the server will deal with the request, or, after some time, the computation is pointless and cancelled. ***Client*** is very easy to model:

$$\begin{aligned}
\boldsymbol{Client} &\overset{def}{=} (think, r_{think}).\boldsymbol{Client\_think} \\
\boldsymbol{Client\_think} &\overset{def}{=} (request, r_{request}).\boldsymbol{Client} \\
&+ (timeout, r_{timeout}).\boldsymbol{Client}
\end{aligned}$$

Servers know two algorithms for dealing with the requests. The first algorithm is very complex and also very fast. Unfortunately, its domain range is quite limited and for certain inputs, an alternative slower algorithm has to be used.

Modelling the ***Server*** component is a little bit trickier. Unfortunately, in GPEPA, when cooperating between groups of components, we cannot model a following situation easily: ***Client*** sends a request, which is answered by a ***Server***. Later, when ***Server*** resolves the request, it would get back to this particular ***Client***. GPEPA simply cannot remember, which ***Client*** and ***Server*** were communicating before. If we tried to model something similar to ***Client*** waiting for a certain action from the ***Server***, then the action might occur in a different ***Server*** during that time and be considered by this ***Client*** as the one awaited, since servers are identical iPEPA components.

Therefore we have to adopt a different approach. Rather than doing the decision process *after* the request arrived, we will set the ***Server*** to one of the algorithms as preparation *before* the request. Although this might appear odd, it makes no difference - the next request will be dealt with either fast or slow algorithm with certain probability. For this we use a ***Solver*** component.

$$\begin{aligned}
\boldsymbol{Solver} &\overset{def}{=} (prepare, \omega_{prepare\_fast} \times \top).\boldsymbol{Solver\_fast} \\
&+ (prepare, \omega_{prepare\_slow} \times \top).\boldsymbol{Solver\_slow} \\
\boldsymbol{Solver\_fast} &\overset{def}{=} (request, r_{solve\_fast}).\boldsymbol{Solver} \\
\boldsymbol{Solver\_slow} &\overset{def}{=} (request, r_{solve\_slow}).\boldsymbol{Solver}
\end{aligned}$$

We will then attach such a ***Solver*** component to our ***Server*** components. A ***Server*** thus first does preparation. With that, it is set to either slow or fast algorithm. Afterwards, it initiates the algorithm by instant randomization of some constants. Then it simply waits for a request. Alternatively, we also consider a case that the server might break during the preparation.

$$
\begin{aligned}
\textbf{\textit{Server}} &\overset{def}{=} (prepare, r_{prepare}).\textbf{\textit{Server\_prepare}} \\
&\quad + (fail, r_{fail}).\textbf{\textit{Server\_fail}} \\
\textbf{\textit{Server\_prepare}} &\overset{def}{=} \mathsf{randomize}.\textbf{\textit{Server\_randomize}} \\
\textbf{\textit{Server\_randomize}} &\overset{def}{=} (request, r_{request}).\textbf{\textit{Server}} \\
\textbf{\textit{Server\_fail}} &\overset{def}{=} (recover, r_{recover}).\textbf{\textit{Server}} \\
\textbf{\textit{Complete\_server}} &\overset{def}{=} \textbf{\textit{Solver}} \underset{\{prepare, request\}}{\bowtie} \textbf{\textit{Server}}
\end{aligned}
$$

The system is then defined as a cooperation between ***Client*** and ***Complete_server*** processes.

$$\textbf{\textit{System}}(N_c, N_s) \overset{def}{=} \mathsf{Clients}\{\textbf{\textit{Client}}[N_c]\} \underset{\{request\}}{\bowtie} \mathsf{Servers}\{\textbf{\textit{Server}}[N_s]\})$$

5.3.2 Example probes

With the defined model, we can finally show few useful passage time calculations. For all these examples we used the same parameters, which were $r_{think} = 0.5$, $r_{request} = 2$, $r_{timeout} = 0.07$, $r_{prepare} = 2$, $r_{fail} = 0.5$, $r_{recover} = 0.2$, $r_{solve\_fast} = 0.7$ and $r_{solve\_slow} = 0.4$. The probabilities for choosing between the algorithms were $\omega_{prepare\_fast} = 0.2$ and $\omega_{prepare\_slow} = 0.8$.

As our first probe we chose, something which is an important indicator of any such system - how long will it, in the steady state, take, till our ***Client*** will get n different calculations. For our examples, we chose n to be 5 and also 10. This is represented in the figures 5.6 and 5.7 respectively. The definition of such a probe follows.

$$
\begin{aligned}
&\textbf{\textit{PM}}_1 \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\qquad \mathsf{observes} \quad \textbf{\textit{Probe}}_l \overset{def}{=} \mathit{think} : \mathsf{begin}, \mathit{request}[n] : \mathsf{end}^{\hookleftarrow} \\
&\qquad \mathsf{where} \quad \mathsf{Clients}\{\textbf{\textit{Client}}[N_c]\} \Longrightarrow \\
&\qquad\qquad \mathsf{Clients}\{(\textbf{\textit{Client}} \underset{*}{\bowtie} \textbf{\textit{Probe}}_l) \wr\wr \textbf{\textit{Client}}[N_c - 1]\} \\
&\qquad \mathsf{in} \quad \textbf{\textit{System}}(N_c, N_s)
\end{aligned}
$$

Similarly, we might be interested in the time, when a ***Client*** will endure n timeouts. We will measure transiently for a change and the probe definition follows. Again, we used $n = 5$ and also $n = 10$, in the figures 5.8 and 5.9 respectively.

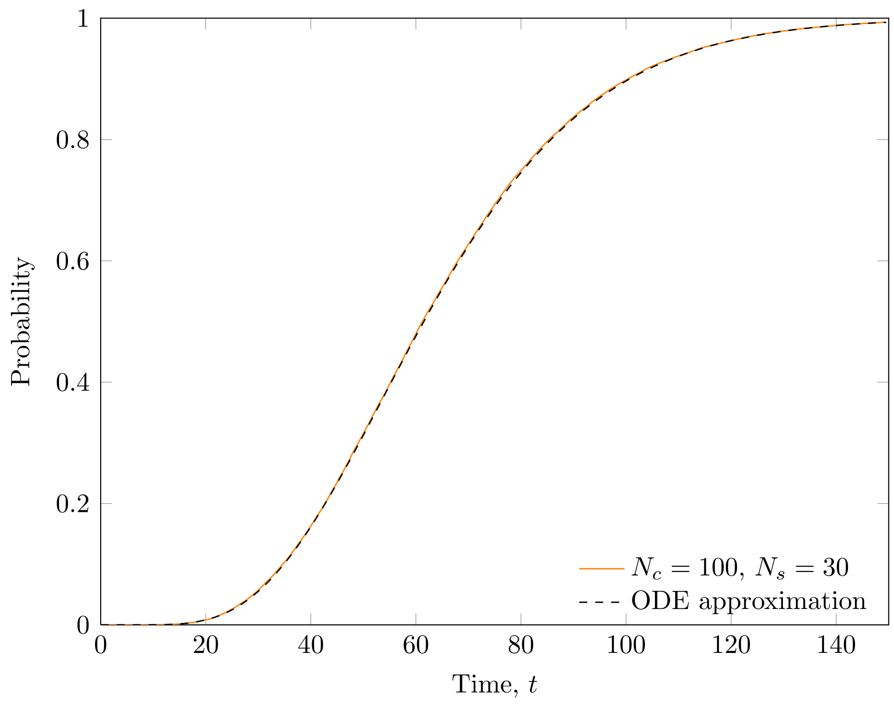

Figure 5.6: PM_1 for $n = 5$

$$
\begin{aligned}
\boldsymbol{PM}_2 &\overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop} \\
&\quad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} think : \mathsf{begin}, timeout[n] : \mathsf{end} \\
&\quad \mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{Client}[N_c]\} \Longrightarrow \\
&\qquad \mathbf{Clients}\{(\boldsymbol{Client} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{Client}[N_c - 1]\} \\
&\quad \mathsf{in} \quad \boldsymbol{System}(N_c, N_s)
\end{aligned}
$$

From these results, we can clearly see that for simple models, the approximation is very precise.

In the next example we will look at the population of ***Server*** components (or ***Complete_server*** components respectively) and determine the time it takes at least half of them to handle n requests. Results for $n = 5$ and $n = 10$ can be then seen in the figures 5.10 and 5.11 respectively. As we can clearly see, the lowest population is not very close to the approximation, at least when compared to greater populations. This simple example proves that fluid flow approximation still has some limitations. In particular, that we still need to work on estimating good population bounds on the rate of convergence.

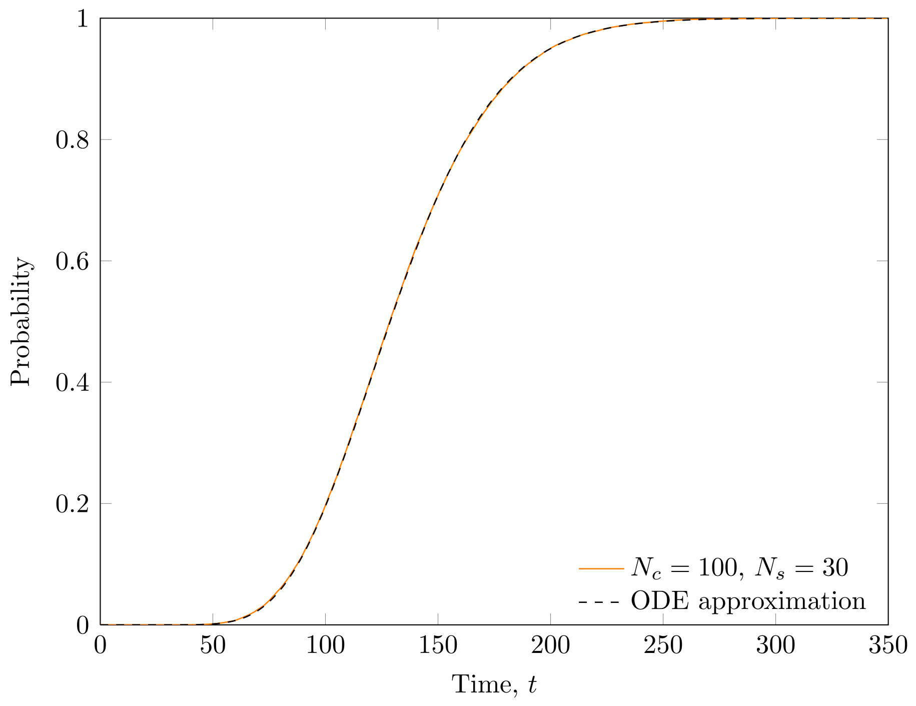

Figure 5.7: PM_1 for $n = 10$

$$\begin{aligned}
&\boldsymbol{PM_3} \overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_s/2] : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe_l} \overset{def}{=} \mathit{request}[n] : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathbf{Servers}\{\boldsymbol{Complete\_server}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Servers}\{(\boldsymbol{Complete\_server} \underset{*}{\bowtie} \boldsymbol{Probe_l})[N_s]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

5.3.3 Complex database system

Our second model is much more complex. Firstly we also have clients, who send transactions. These are handled by a scheduling system, which checks the requests and the valid ones are added to transactions buffers. Later, they are dealt with by some of the servers. Although it might sounds as a complicated system, in iGPEPA it is surprisingly easy to model.

The ***Client*** component is the simplest. It decides on a transaction and then simply requests it to be processed. Sometimes, it decides on a batch of transactions at once.

$$\begin{aligned}
&\boldsymbol{Client} && \overset{def}{=} (think, r_{think}).\boldsymbol{Client\_think}\\
&\boldsymbol{Client\_think} && \overset{def}{=} (request, r_{request}).\boldsymbol{Client}\\
& && + (request, r_request_batch).\boldsymbol{Client}_think
\end{aligned}$$

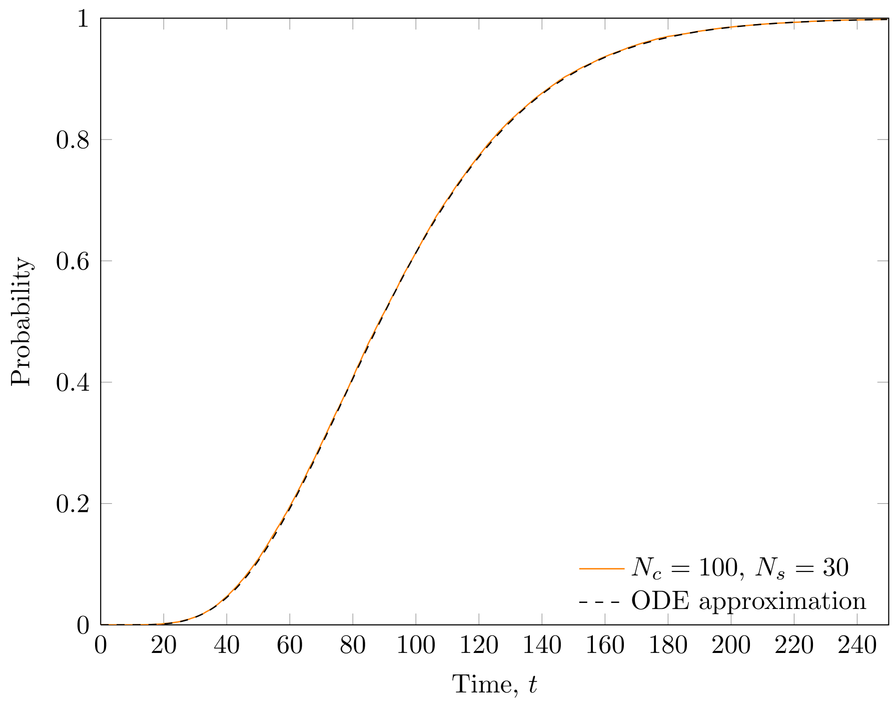

Figure 5.8: PM_2 for $n = 5$

We then have multiple transaction buffers. Each of these, before adding a request to its queue, checks, whether it is a valid transaction. If it is, it proceeds, and adds it to the buffer, when there is a free space. If not, it waits. That is the reason why optimally setup system uses multiple buffers.

$$\begin{aligned}
\boldsymbol{Scheduler} &\overset{def}{=} (request, r_{request}).\boldsymbol{Scheduler\_request} \\
\boldsymbol{Scheduler\_request} &\overset{def}{=} (check, r_{check\_success}).\boldsymbol{Scheduler\_check} \\
&+ (check, r_{check\_fail}).\boldsymbol{Scheduler} \\
\boldsymbol{Scheduler\_add} &\overset{def}{=} (add, r_{add}).\boldsymbol{Scheduler}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{Buffer}_0 &\overset{def}{=} (add, \top).\boldsymbol{Buffer}_1 \\
\boldsymbol{Buffer}_i &\overset{def}{=} (add, \top).\boldsymbol{Buffer}_{i+1} \\
&+ (serve, \top).\boldsymbol{Buffer}_{i-1} \\
\boldsymbol{Buffer}_B &\overset{def}{=} (serve, \top).\boldsymbol{Buffer}_{B-1} \\
\boldsymbol{ManagedBuffer} &\overset{def}{=} \boldsymbol{Scheduler} \underset{\{add\}}{\bowtie} \boldsymbol{Buffer}_0
\end{aligned}$$

Finally, our ***Server***. Each of them uses a special ***DatabaseDriver*** subsystem, which needs to be running, in order to allow the ***Server*** to handle a transaction. If there is an error in the ***DatabaseDriver*** during initialisation, it will cause a system restart. Similarly, if the ***Server*** itself restarts after overloading, we have to restart its ***DatabaseDriver*** too.

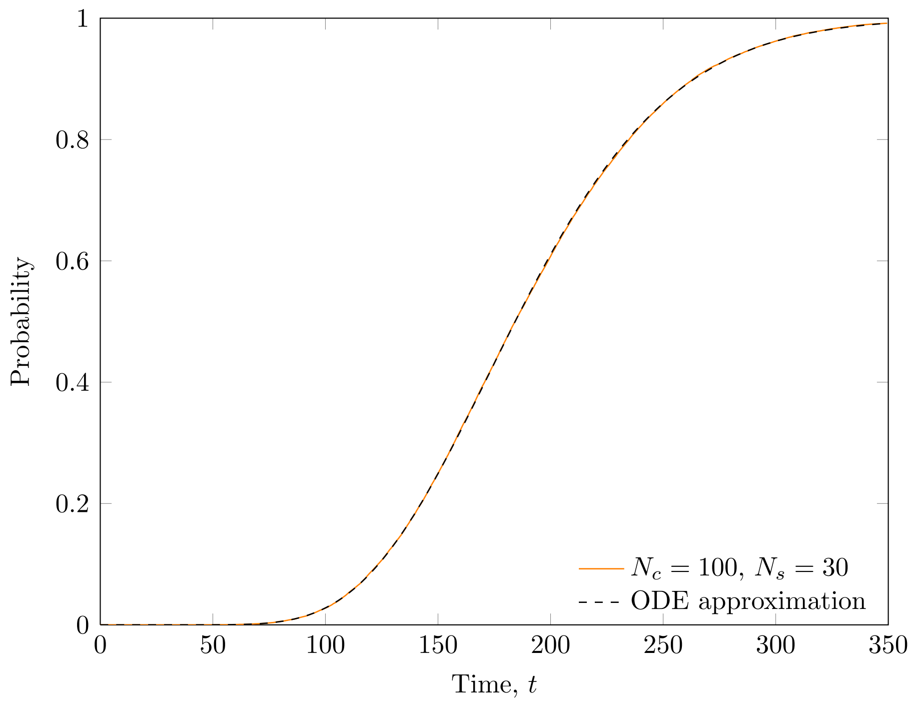

Figure 5.9: PM_2 for $n = 10$

$$\begin{aligned}
\boldsymbol{DatabaseDriver} &\overset{def}{=} (\mathit{initialise}, \omega_{\mathit{initialise\_success}} \times \top).\boldsymbol{DatabaseDriver\_initialise} \\
&+ (\mathit{initialise}, \omega_{\mathit{initialise\_fail}} \times \top).\boldsymbol{DatabaseDriver\_fail} \\
\boldsymbol{DatabaseDriver\_fail} &\overset{def}{=} (\mathit{reset}, r_{\mathit{reset}}).\boldsymbol{DatabaseDriver} \\
\boldsymbol{DatabaseDriver\_initialise} &\overset{def}{=} (\mathit{serve}, \top).\boldsymbol{DatabaseDriver\_initialise} \\
&+ (\mathit{reset}, \top).\boldsymbol{DatabaseDriver}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{Server} &\overset{def}{=} (\mathit{initialise}, r_{\mathit{initialise}}).\boldsymbol{Server\_initialise} \\
\boldsymbol{Server\_initialise} &\overset{def}{=} (\mathit{serve}, r_{\mathit{serve}}).\boldsymbol{Server\_initialise} \\
&+ (\mathit{reset}, \top).\boldsymbol{Server} \\
&+ (\mathit{overload}, r_{\mathit{overload}}).\boldsymbol{Server\_overload} \\
\boldsymbol{Server\_overload} &\overset{def}{=} \mathsf{reset}.\boldsymbol{Server} \\
\boldsymbol{CompleteServer} &\overset{def}{=} \boldsymbol{Server} \underset{\{\mathit{initialise},\mathit{serve},\mathit{reset}\}}{\bowtie} \boldsymbol{DatabaseDriver}
\end{aligned}$$

The complete system is then defined as a cooperation between ***Client***, ***ManagedBuffer*** and ***Server*** components.

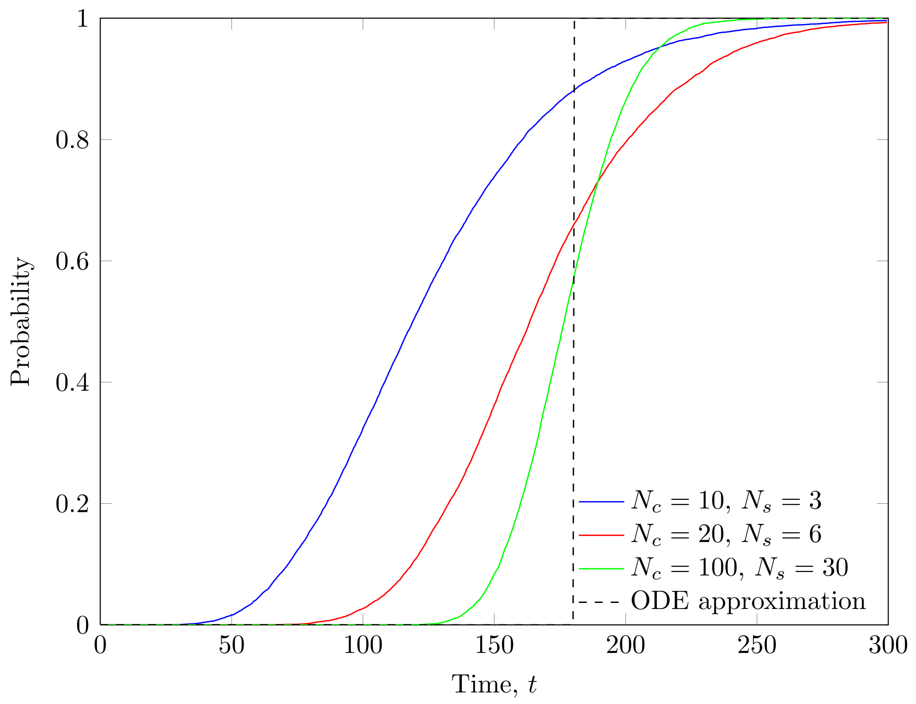

Figure 5.10: PM_3 for $n = 5$

$$\begin{aligned}\boldsymbol{System}(N_c, N_b, N_s) &\overset{def}{=} \quad (\textbf{Clients}\{\boldsymbol{Client}[N_c]\} \underset{\{request\}}{\bowtie} \textbf{Buffers}\{\boldsymbol{ManagedBuffer}[N_b]\}) \\ &\qquad \underset{\{serve\}}{\bowtie} \textbf{Servers}\{\boldsymbol{CompleteServer}[N_s]\}\end{aligned}$$

5.3.4 Example probes

Now we can demonstrate some useful metrics on our system. For this section, we used the parameters defined as $r_{think} = 0.1, r_{request} = 0.5, r_{request\_batch} = 0.3, r_{check\_success} = 0.6, r_{check\_fail} = 0.3, r_{add} = 0.7, r_{serve} = 0.2, r_{init} = 0.6, r_{overload} = 0.1$ and $r_{reset} = 1$. The probabilities of successful or unsuccessful initialisation of ***DatabaseDriver*** were $\omega_{init\_success} = 0.9$ and $\omega_{init\_fail} = 0.1$. We defined $B = 2$, so that buffers could hold up to 2 transactions.

The first example, PM_4, measures how long it takes a ***Client*** to send 10 requests, given it has sent at least a batch of two requests already. We take this measurement in the steady state. (Figure 5.12)

$$\begin{aligned}&\boldsymbol{PM}_4 \overset{def}{=} \textsf{begin : start, end : stop}^{\hookleftarrow} \\ &\qquad \textsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} (request[2]/think) : \textsf{begin}, request[10] : \textsf{end}^{\hookleftarrow} \\ &\qquad \textsf{where} \quad \textbf{Clients}\{\boldsymbol{Client}[N_c]\} \Longrightarrow \\ &\qquad\qquad \textbf{Clients}\{(\boldsymbol{Client} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr \boldsymbol{Client}[N_c - 1]\} \\ &\qquad \textsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)\end{aligned}$$

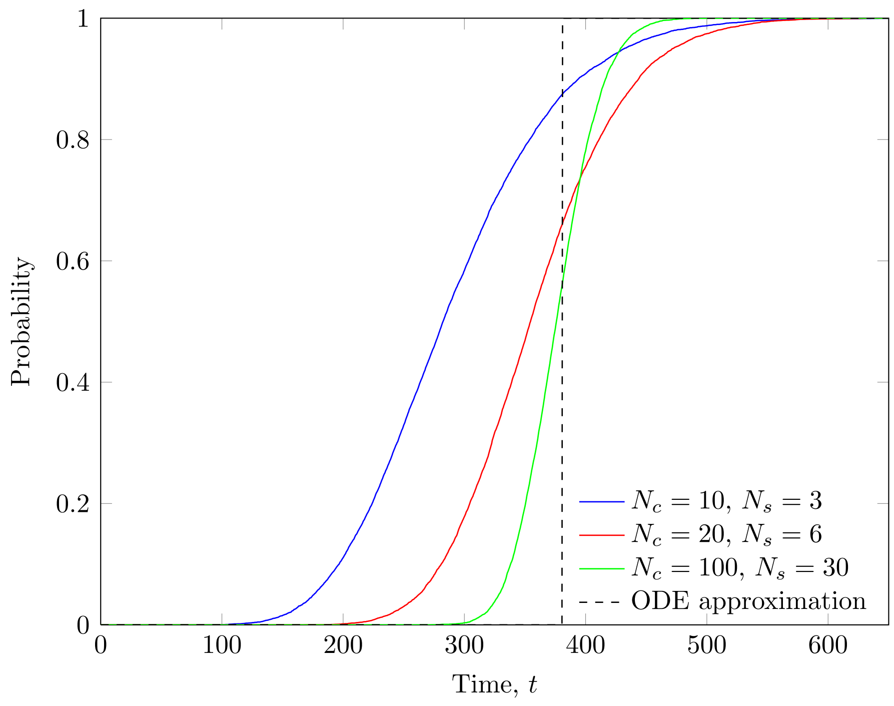

Figure 5.11: PM_3 for $n = 5$

PM_5 measures the time, since a ***CompleteServer*** initialises till it resets 12 times. (Figure 5.13)

$$\begin{aligned}
&\boldsymbol{PM}_5 \stackrel{\mathit{def}}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \stackrel{\mathit{def}}{=} \mathit{initialise} : \mathsf{begin}, \mathit{reset}[12] : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathrm{Servers}\{\boldsymbol{CompleteServer}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathrm{Servers}\{(\boldsymbol{CompleteServer} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{CompleteServer}[N_s - 1]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

In the PM_6, we expect 80% of the servers to 8 times overload and also 8 times handle a request. (Figure 5.14)

$$\begin{aligned}
&\boldsymbol{PM}_6 \stackrel{\mathit{def}}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_s * 0.8] : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \stackrel{\mathit{def}}{=} (\mathit{overload}[8]; \mathit{serve}[8]) : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathrm{Servers}\{\boldsymbol{CompleteServer}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathrm{Servers}\{(\boldsymbol{CompleteServer} \underset{*}{\bowtie} \boldsymbol{Probe}_l)[N_s]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

PM_7 measures the time to add 5 transactions into a ***ManagedBuffer***. However, we start the

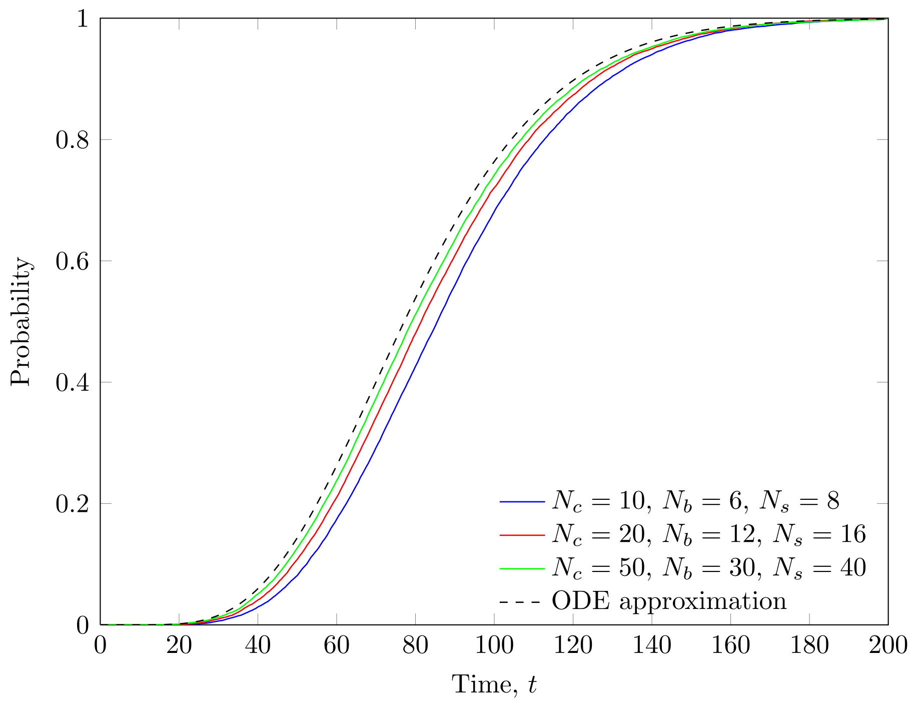

Figure 5.12: PM_4

measurement in the equilibrium, wait until we send a request and a successful add operation is performed, not necessarily on this request. (Figure 5.15)

$$\begin{aligned}
&\boldsymbol{PM_7} \overset{\text{def}}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe_l} \overset{\text{def}}{=} (request, ((check, add)/request)) : \mathsf{begin}, add[5] : \mathsf{end}^{\hookleftarrow}\\
&\qquad \mathsf{where} \quad \mathbf{Buffers}\{\boldsymbol{ManagedBuffer}[N_b]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Buffers}\{(\boldsymbol{ManagedBuffer} \underset{*}{\bowtie} \boldsymbol{Probe_l}) \wr\wr \boldsymbol{ManagedBuffer}[N_b - 1]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

PM_8 measures the time from initialisation of a ***CompleteServer*** to serving a request 3 times without resetting in between. (Figure 5.16)

$$\begin{aligned}
&\boldsymbol{PM_8} \overset{\text{def}}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe_l} \overset{\text{def}}{=} initialise : \mathsf{begin}, (serve[3]/\boldsymbol{reset}) : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathbf{Servers}\{\boldsymbol{CompleteServer}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Servers}\{(\boldsymbol{CompleteServer} \underset{*}{\bowtie} \boldsymbol{Probe_l}) \wr\wr \boldsymbol{CompleteServer}[N_s - 1]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

Now in PM_9 we are interested in the time it takes 95% of ***Client*** components to send 20

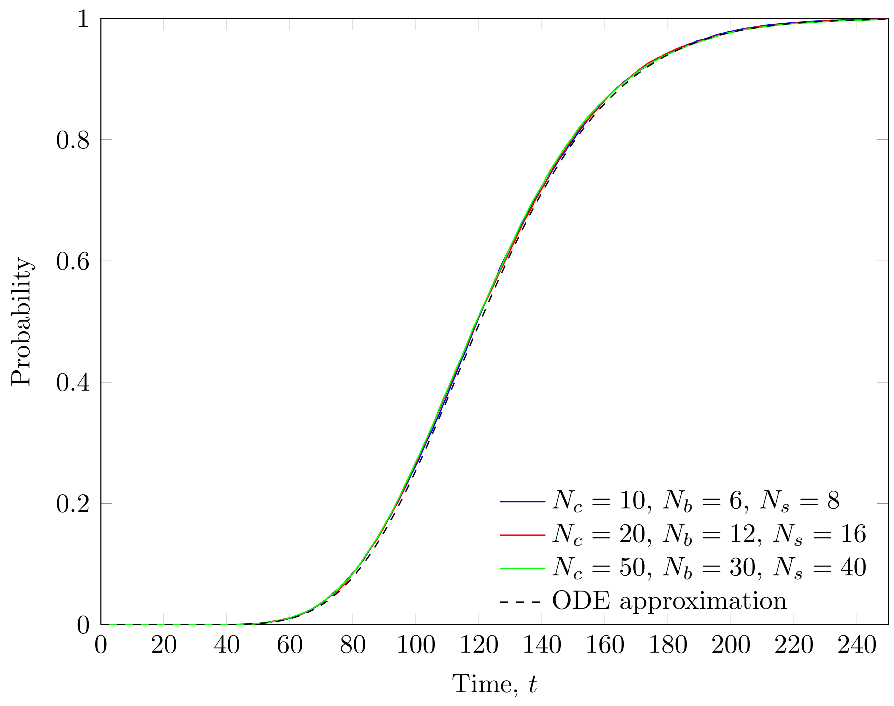

Figure 5.13: PM_5

requests. (Figure 5.17)

$$\begin{aligned}
&\boldsymbol{PM_9} \overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_c * 0.95] : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe_l} \overset{def}{=} \mathit{request}[20] : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{Client}[N_c]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Clients}\{(\boldsymbol{Client} \underset{*}{\bowtie} \boldsymbol{Probe_l})[N_c]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

Probe PM_{10} demonstrates the combinatory capabilities of the probes language. In the steady state, we expect a server to initialise and serve, and also reset twice, while not overloading (so that we count only resets from the ***DatabaseDriver***). Additionally, we repeat this three times. (Figure 5.18)

$$\begin{aligned}
&\boldsymbol{PM_{10}} \overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe_l} \overset{def}{=} \epsilon : \mathsf{begin}, (((\mathit{initialise}, \mathit{serve}); \boldsymbol{reset}[2]), \mathit{overload})[3] : \mathsf{end}^{\hookleftarrow}\\
&\qquad \mathsf{where} \quad \mathbf{Servers}\{\boldsymbol{CompleteServer}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Servers}\{(\boldsymbol{CompleteServer} \underset{*}{\bowtie} \boldsymbol{Probe_l}) \wr\wr \boldsymbol{CompleteServer}[N_s - 1]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}$$

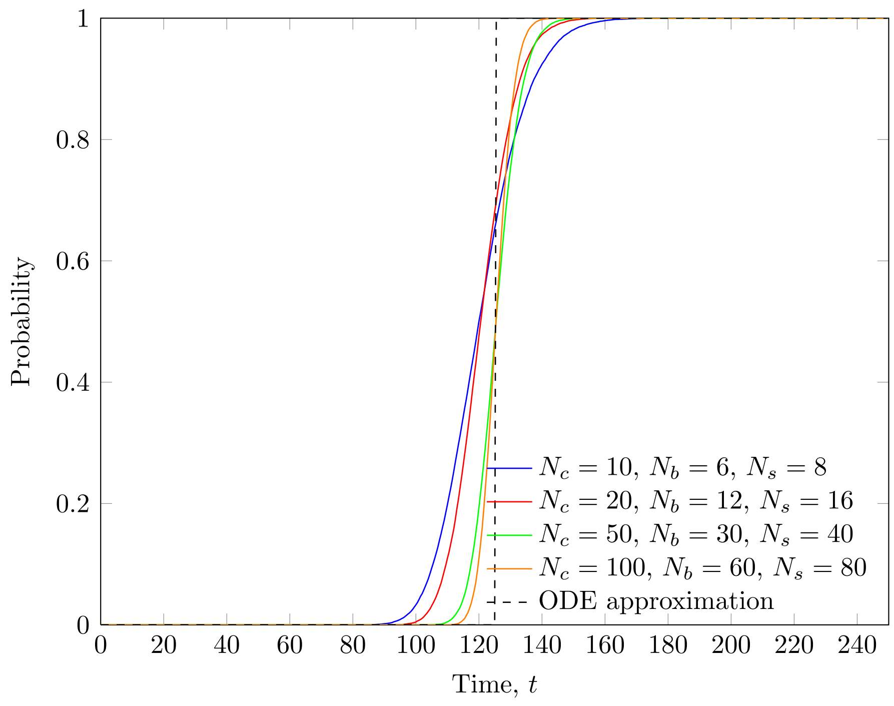

Figure 5.14: PM_6

PM_{11} simply measures when will a server handle its first 12 transactions. (Figure 5.19)

$$
\begin{aligned}
\boldsymbol{PM}_{11} &\overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop} \\
&\mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} eE : \mathsf{begin}, request[12] : \mathsf{end} \\
&\mathsf{where} \quad \mathbf{Clients}\{\boldsymbol{Client}[N_c]\} \Longrightarrow \\
&\qquad \mathbf{Clients}\{(\boldsymbol{Client} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr \boldsymbol{Client}[N_c - 1]\} \\
&\mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}
$$

Next in the probe PM_{12} we will measure the time until half of the buffers successfully add 25 transactions. (Figure 5.20)

$$
\begin{aligned}
\boldsymbol{PM}_{12} &\overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_b/2] : \mathsf{stop} \\
&\mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} add[25] : \mathsf{end} \\
&\mathsf{where} \quad \mathbf{Buffers}\{\boldsymbol{ManagedBuffer}[N_b]\} \Longrightarrow \\
&\qquad \mathbf{Buffers}\{(\boldsymbol{ManagedBuffer} \underset{*}{\bowtie} \boldsymbol{Probe}_l)[N_b]\} \\
&\mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}
$$

With PM_{13} we are interested in the time until a buffer adds or pushes out 20 transactions, given that it has already seen a request in equilibrium. (Figure 5.21)

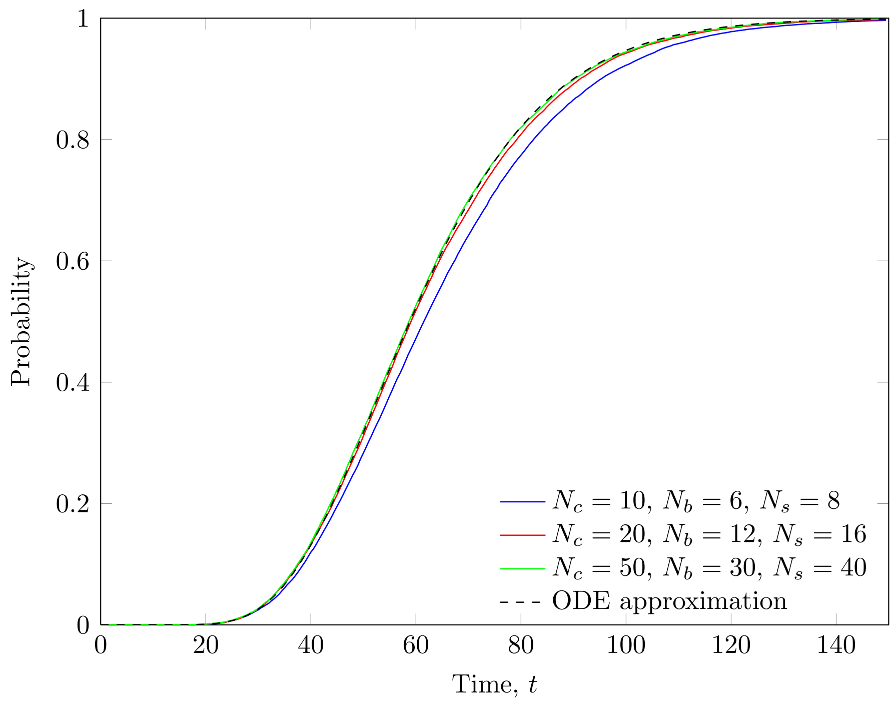

Figure 5.15: PM_7

$$
\begin{aligned}
\boldsymbol{PM}_{13} &\overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop}^{\hookleftarrow} \\
&\quad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} (request) : \mathsf{begin}, (add|serve)[20] : \mathsf{end}^{\hookleftarrow} \\
&\quad \mathsf{where} \quad \mathbf{Buffers}\{\boldsymbol{ManagedBuffer}[N_b]\} \Longrightarrow \\
&\qquad \mathbf{Buffers}\{(\boldsymbol{ManagedBuffer} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{ManagedBuffer}[N_b - 1]\} \\
&\quad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}
$$

Here in the probe PM_{14} we query the system for the time, when a buffer adds and also removes transactions at least 8 times. (Figure 5.22)

$$
\begin{aligned}
\boldsymbol{PM}_{14} &\overset{def}{=} \mathsf{begin} : \mathsf{start}, \mathsf{end} : \mathsf{stop} \\
&\quad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} \epsilon : \mathsf{begin}, (add; serve)[8] : \mathsf{end} \\
&\quad \mathsf{where} \quad \mathbf{Buffers}\{\boldsymbol{ManagedBuffer}[N_b]\} \Longrightarrow \\
&\qquad \mathbf{Buffers}\{(\boldsymbol{ManagedBuffer} \underset{*}{\bowtie} \boldsymbol{Probe}_l) \wr\wr \boldsymbol{ManagedBuffer}[N_b - 1]\} \\
&\quad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}
$$

The last example, PM_{15} shows, when will at least 60% of the servers handle 5 transactions without resetting in between. (Figure 5.23)

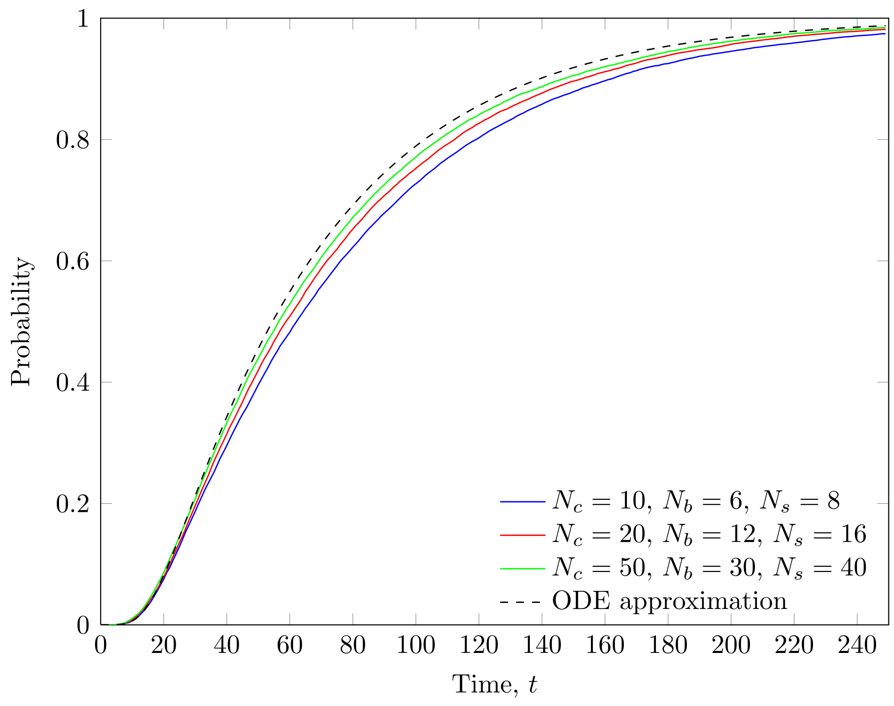

Figure 5.16: PM_8

$$
\begin{aligned}
&\boldsymbol{PM}_{15} \overset{def}{=} \epsilon : \mathsf{start}, \mathsf{end}[N_s * 0.6] : \mathsf{stop}\\
&\qquad \mathsf{observes} \quad \boldsymbol{Probe}_l \overset{def}{=} (serve[5]/reset) : \mathsf{end}\\
&\qquad \mathsf{where} \quad \mathbf{Servers}\{\boldsymbol{CompleteServer}[N_s]\} \Longrightarrow\\
&\qquad\qquad \mathbf{Servers}\{(\boldsymbol{CompleteServer} \underset{*}{\bowtie} \boldsymbol{Probe}_l)[N_s]\}\\
&\qquad \mathsf{in} \quad \boldsymbol{System}(N_c, N_b, N_s)
\end{aligned}
$$

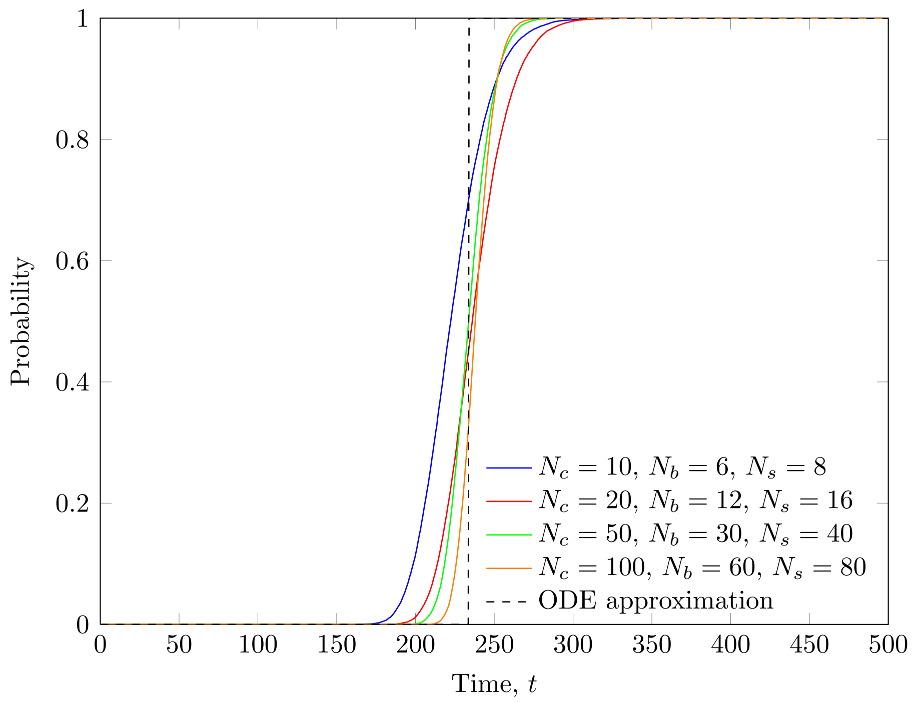

Figure 5.17: PM_9

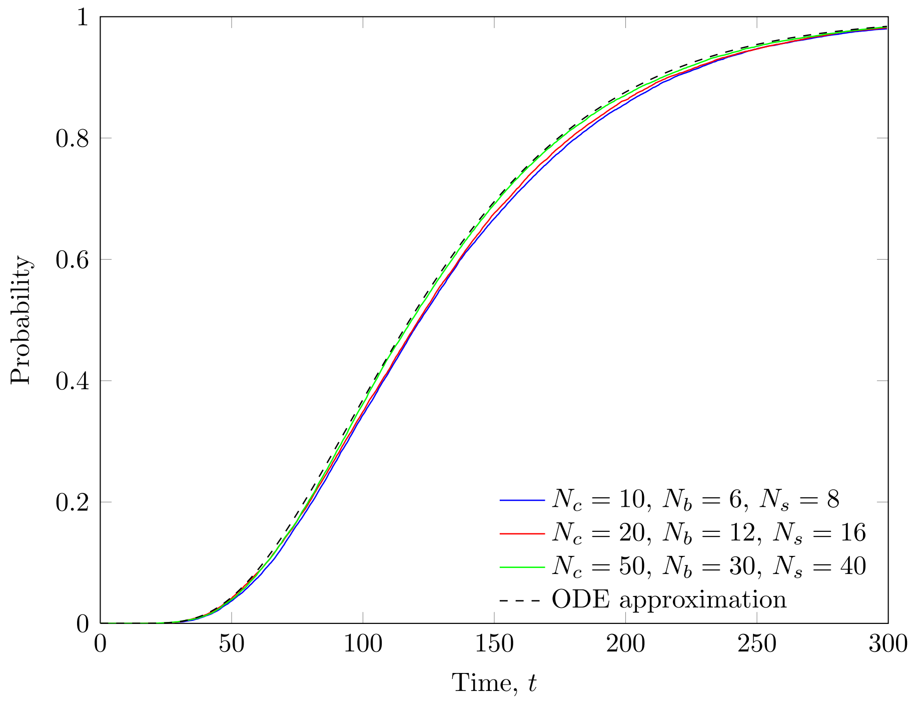

Figure 5.18: PM_{10}

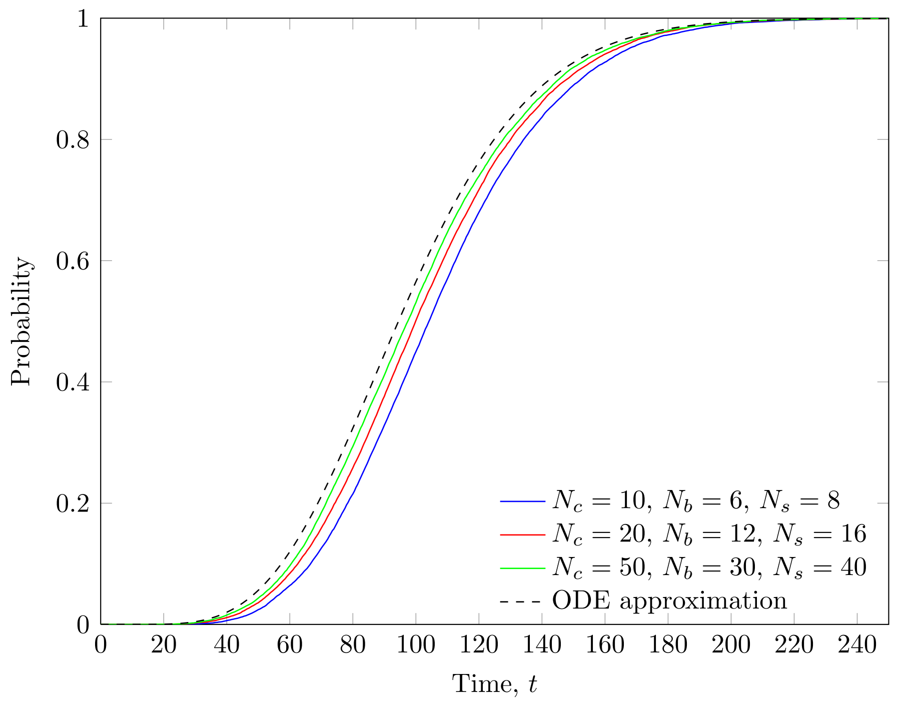

Figure 5.19: PM_{11}

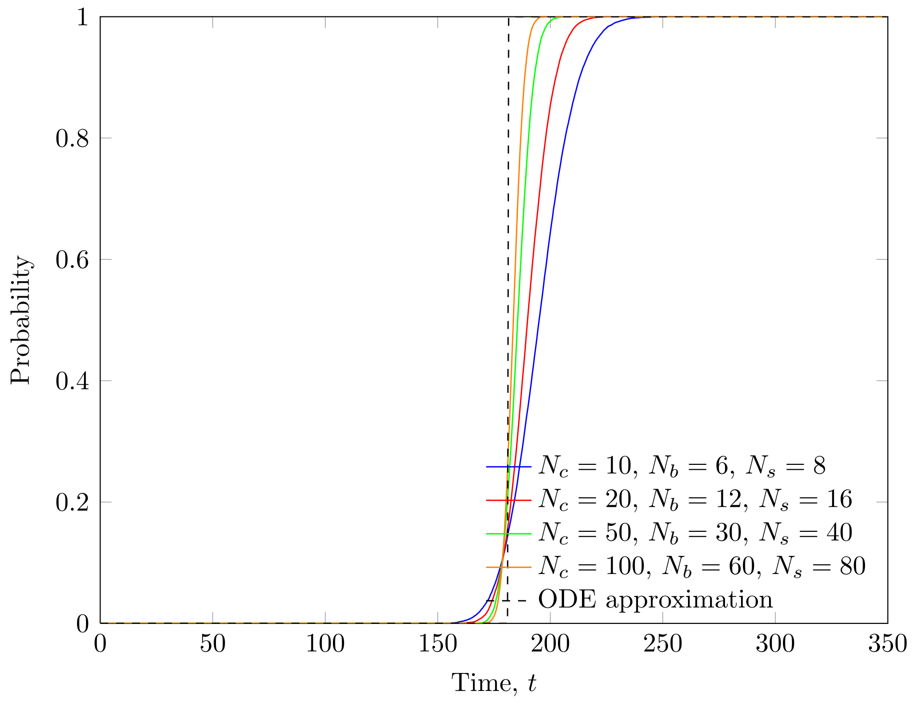

Figure 5.20: PM_{12}

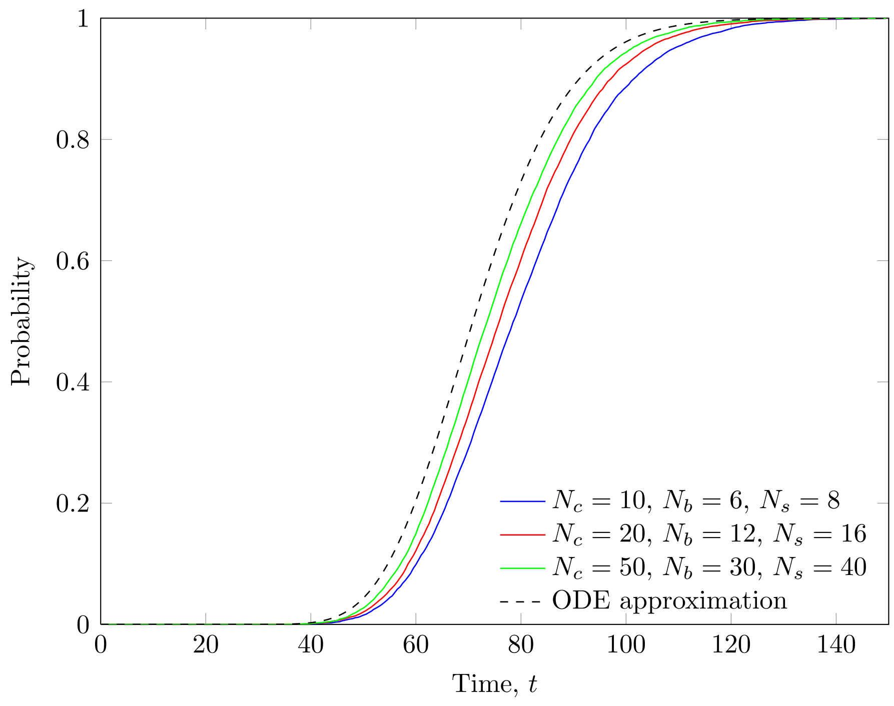

Figure 5.21: PM_{13}

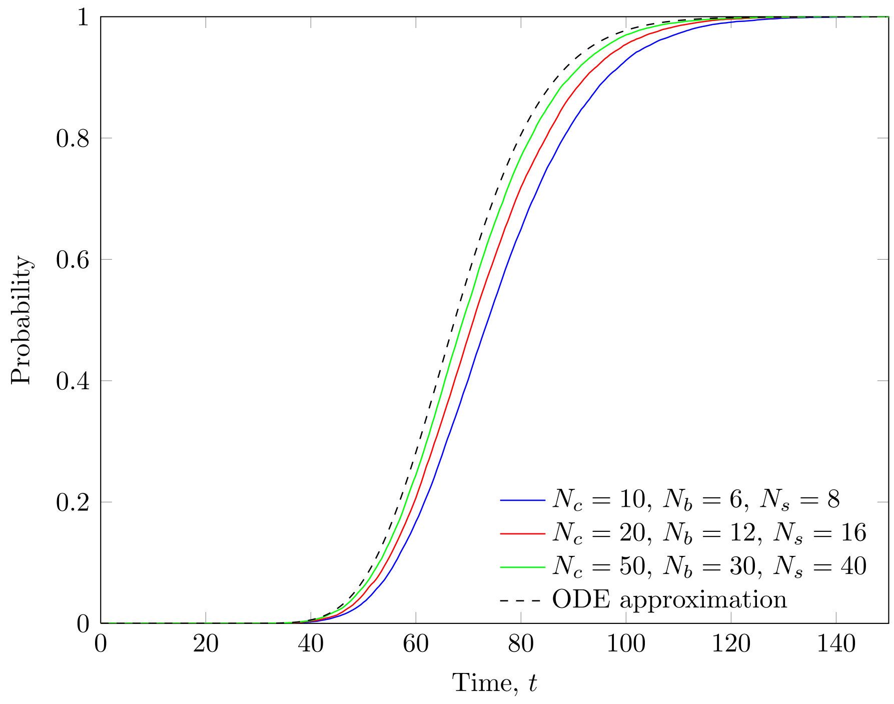

Figure 5.22: PM_{14}

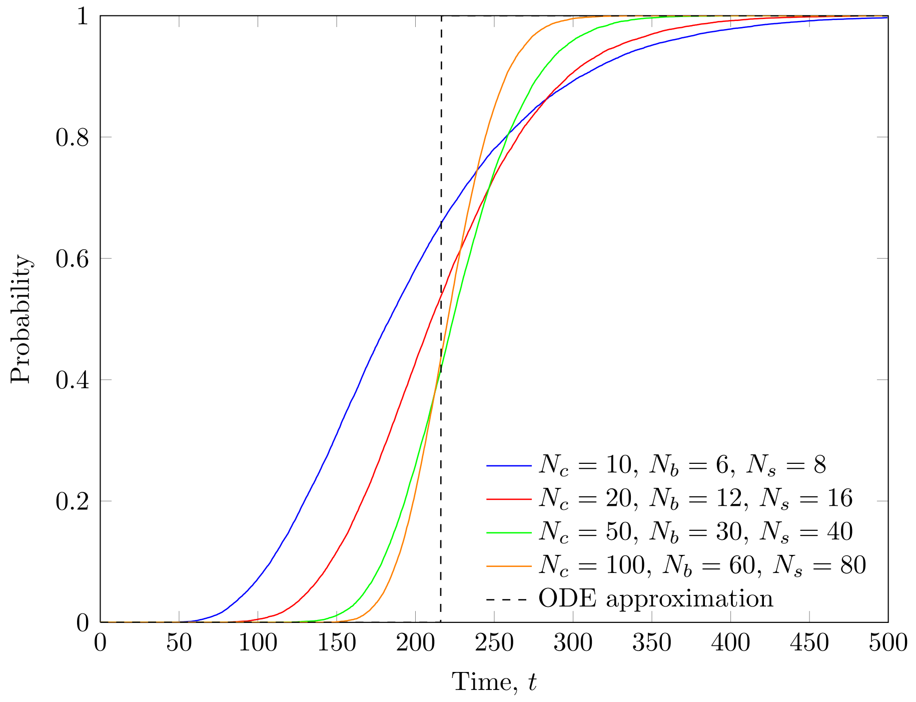

Figure 5.23: PM_{15}

5.4 Conclusion

We have shown how the Unified Stochastic Probes formalism for computing complex passage time queries can be more accessible by combining it with the graphical Performance Trees formalism. We have also proved this combination subsumes the originally proposed Performance Trees way for computing passage-time densities (the Appendix A).

These concepts were demonstrated to be feasible by a full working implementation of the Unified Stochastic Probes formalism along with passage-time calculations in both simulation and fluid flow approximation modes. This was achieved by adding the support of the Probes formalism to the open-source GPAnalyser software and also extending its implementation of the GPEPA language to support passive, weighted passive and immediate actions. Furthermore, the complexity of the tractable models was significantly increased by using C++ and JNI rather than pure Java dynamic compilation technique.

Finally, we have demonstrated the power of our implementation by comparing its computations with the manually hand-crafted results from [24] and by analysing our own iGPEPA models. We have also shown a range of new examples for passage time calculations to reinforce the concepts.

5.5 Future work

During the course of this project, there have been multiple areas discovered, which deserve a further research. We have also identified several possible extensions to our implementation, which could increase the usefulness of the Probes formalism.

- We plan to introduce passage time computations using the *higher-order moments*.
- We would like to explore the idea of spatially extending the Unified Stochastic Probes formalism and provide a simple way for using it with spatially-enabled process algebras, such as Bio-PEPA ([17]) or MASSPA ([20]).
- We are looking into *functional rates* in PEPA so that we can support global probe predicates in the simulation mode.
- Global probes in the fluid flow approximation mode need formal reviewing and developing their permitted grammar beyond the current limited scope.
- Future work includes an implementation of a tool for the newly proposed graphical syntax of the Unified Stochastic Probes formalism.
- We plan to increase the support for immediate actions in our implementation to *non-deterministic signalling paths* from the non-initial states and thus allowing for specifying any well-behaved components.

Appendices

Appendix A

Translation of PTD node to a Probe PTD node

In this appendix, we will show how even the weakest kind of Unified Stochastic Probes can achieve the same as the PTD node originally defined in [30]. Since we will use a very small subset of the operations permitted in Unified Stochastic Probes formalism, we can conclude that behavioural-oriented probes subsume and surpass the original definition. Furthermore, they are more intuitive to define.

The weakest kind of Unified Stochastic Probe is a global probe which observes a system with no local probes attached. With this type of Probe we can only monitor the overall, global, behaviour of the system and its passages. In this section, we will assume two simple extensions of global probe predicates here - testing for *impulse* and *accumulated rate rewards*. For impulse rewards we will use *action-counting ODEs*, as proven in [22, Section 5.4.1]. Current GPAnalyser version supports both of these reward types, therefore using these in predicates is not unrealistic in the future versions. In our case, impulse rewards track how many times an action has been executed in the model. Accumulated rate rewards track a quantity of interest as some function of the system over time, e.g. fuel consumption in a motor model.

Before we demonstrate translation of our first Performance Query into the probes language, we will need some model modifications to accommodate for some functional requirements of the originally defined PTD node.

A.1 Model modifications

In order to accommodate for excluded actions and states from the original PTD node, we will make some simple modifications to the model.

Every Prefix with an excluded action α leading to a state s will be substituted by an α self-loop with the same rate/weight. This is to ensure the model will not use any excluded action during the passage time evaluation. This change will have no probabilistic impact on the overall run of the model. We did preserve all the actions and their rate/parameters in all the reachable states, which means no rates, or action weights will be altered.

We can also prove, that by keeping these dummy self-loops, we do not alter the race between the available transitions. There are two cases:

- An allowed transition wins the race. In this case, the overall behaviour was not affected by any changed self-loops.
- An excluded transition wins the race.

The second case is trickier to prove. Suppose the excluded transitions wins the race after a time interval t since the start of the race. Then we end up in the same state where we started, since we used a self-loop and a new race starts right away. The probability of an allowed action α in a time period s occurring is then $P(S < s)$.

For comparison, lets assume the excluded action did not win the first race and the original race continued until this α is performed. This is the case we are interested in. Since we know, that in time t no action has occurred, we can write the probability of action α occurring in the next time period s after t as $P(S < s + t | S \geq t)$.

Now we can use the *Markov property* of *CTMC*, since the structure underlying a GPEPA model is a CTMC and the transitions are Markov Processes. By this property the aforementioned probabilities are equal. From that, we can conclude the passage time probabilities of interest have not been altered by this simple change.

We can use the same tactics for all excluded states - we substitute every Prefix leading to an excluded state by a self-loop Prefix with the same action and rate/weight. Again this did not affect any probabilities or rates of actions. We can use the same proof as for excluded actions to show that the passage time of interest has not been affected.

A.2 Probe generation

The original PTD definition allowed specifying multiple different starting states of the complete system. Because of the way the GPAnalyser works, we are only interested in a single starting state. For other starting states we can just alter the overall model. We can then use the function specified in (A.2) to obtain the final CDF.

While global probe predicates do not support conjunction operation in their original definition, it can be emulated by stacking up the predicates. For *predicate* 1 and *predicate* 2, the conjunction would be defined as $\{predicate\ 1\}\{predicate\ 2\}$, which follows from the global probe predicates grammar.

For the other requirements (included states, included actions, target states, rewards) we can use the global probe predicate logic. We are essentially interested in global probes of the form

$$\epsilon : start, \{end\ condition(s)\}\epsilon : stop \tag{A.1}$$

since these measure the time until the whole system reaches a state satisfying certain conditions.

The reaching of any of the target states can be easily detected by a global probe predicate. For each of the target states, we have a set of statements determining how many particular system components should be in what component state. For example, returning back to the model from Subsection 2.4.3, we can be interested in a state system, where we have N_c ***Consumer_get*** and N_p ***Producer*** components. This can be expressed by a global probe predicate $\{\textbf{\textit{Consumer\_get}} = N_c\}\{\textbf{\textit{Producer}} = N_p\}$.

If we have n target overall states, we can in this manner generate multiple probes, each for one of them. Then we join the obtained CDFs with a function defined as

$$\forall t \in \mathbb{R}_+ : f(t) = \max x_1, ..., x_n | x_i = CDF_i(t) \tag{A.2}$$

where $CDF_i(t)$ represents the numerical value of the CDF generated from i-th probe at time t. The intuitive explanation is, we are interested in the maximal density probability at each time.

For included actions, we can use the extended predicate logic. For each included action α, we add one predicate with $\{\alpha = n\}$, where n is the number of times this action is included.

For accumulated rewards we can again use the extended predicate logic and simply test these with a simple predicate in conjunction with the others so far.

Included states are bit trickier. Lets assume we have n different included states. First step is to generate a probe from the start state to all of them as if they were the target states, but we do not use the other conditions yet (included actions and rewards). Since this is a Global Probe culminating in the fluid-flow mode to a point mass approximation, we can use the time of the probability turning into 1 directly. We than pick the minimal time obtained from these results and take the state it is associated with it as our new start state. We than generate probes for all the included states starting from this new start state, except for the one, which has won the previous race. We evaluate the next winner in a similar manner as the first one and continue with this process until there are no included states left. Only then we will go on to evaluate race into the end states as previously described and use the other conditions in the predicate (included actions and rewards). In the end, we need to add the winning accumulated times to get the resulting time of the point-mass approximation.

A.3 Conclusion

We have shown we are able to use probes to emulate the originally defined PTD node. Even though in this case probes' use is inconvenient, since probes do not directly work with system states, except for the global probe predicates, it is still possible to use them as a substitute. However, as previously stated, the PTD node was originally proposed in this manner because there was no other viable alternative. In truth, we are rarely interested in the states of the system when considering its behaviour over the time. We are more interested in the actual actions executed by an individual or all components. Probes are therefore far superior in both intuitiveness and expressive power, since they easily allow also observing either transient or steady-state individual passage times, unlike the original PTD node. Furthermore, they also provide the option to specify the ordering and necessity of the executed actions, which is impossible with a PTD node.

Appendix B

User guide

In this appendix we present the useful program options and the newly introduced syntax for GPAnalyser. We end this guide with some examples of both models and probes.

B.1 Program options

GPAnalyser supports a range of extra options. For our purposes, we introduce only some interesting ones.

- Option *noGUI* specifies, that all the graphical output, including CDF from Probes, should be supressed. It can be used for automated runs or runs in a cluster.
- Option *cpp* chooses our C++/JNI (the Section 4.8) compilation rather than the default Java one (currently for Probes only). It requires *g++* to be set up on the host computer and accessible from the command line. This option takes one argument - the directory, where all the generated files will be stored.

B.2 iGPEPA syntax

Here we present syntax we use in GPAnalyser. It is resembling of the traditional iGPEPA syntax, however there are minor differences. These were introduced to simplify writing of the models and make models clearer. Sometimes, it was necessary (synchronization symbol from PEPA).

- Prefix - uses the same syntax as in iGPEPA, `(alpha, ra).Component`. It says action *alpha* is available, with rate "ra" and will result in the ***Component*** component.
- Passive prefix - comes in two variants. First is `(alpha, T, wa).Component`. It specifies *alpha* as a passive action with weight "wa". The second is simply `(alpha, T).Component`, which is a shorthand for `(alpha, T, 1).Component`.
- Immediate prefix - again there are two flavours. First is `[alpha, wa].Component`. It specifies immediate alpha action with weight "wa". The second form is simply `alpha.Component`,

which is a shorthand for `[alpha, 1].Component`. Since we only support deterministic signalling paths, the second flavour is currently preferred and the weights are ignored.

- Choice - we also use the "+" (addition) operator. `(alpha, ra).ComponentA + (beta, rb).ComponentB` will either result in ***ComponentA*** or in ***ComponentB*** by racing the *alpha* and *beta* in the usual manner.
- Cooperation - we use angle brackets. For example, `ComponentA<alpha>ComponentB` specifies a cooperation between ***ComponentA*** and ***ComponentB*** synchronized on action *alpha*.
- Parallel cooperation between multiple components of same type - we use the same syntax as iGPEPA. For example, `Component[nc]` means *nc* components of type ***Component*** cooperating independently in parallel.
- Group - we use the same syntax as iGPEPA. For instance, **Group1**{`Component[nc]`} specifies **Group1** as a group uniquely labelling *nc* particular ***Component*** processes cooperating in parallel.
- Groups cooperation - for synchronized cooperation we use the same syntax as for PEPA cooperation. That means **Group1**{`Component`}`<alpha>`**Group2**{`Component`} specifies cooperation between two groups of components synchronized on action *alpha*. For unsynchronized cooperation, simple "`|`" suffices, so **Group1**{`Component`} `|` **Group2**{`Component`}.

B.2.1 Example

Returning to our example from the Subsection 5.3.1, it's syntax would then be represented as follows. The first line illustrates how to define constants.

```
nc = 5;

Client                = (think, r_think).Client_think;
 Client_think         = (request, r_request).Client;
                      + (timeout, r_timeout).Client;

Solver                = (prepare, T, prepare_fast).Solver_fast;
                      + (prepare, T, prepare_slow).Solve_slow;

 Solver_fast          = (request, r_solve_fast).Solver;
 Solver_slow          = (request, r_solve_slow).Solver;

Server                = (prepare, r_prepare).Server_prepare;
                      + (fail, r_fail).Server_fail;
 Server_prepare       = randomize.Server_randomize;
 Sever_randomize      = (request, T).Server
                      + (clear, rcl).Producer;

CompleteServer        = Solver<prepare, request>Server;

Clients{Client[nc]}<request>Servers{CompleteServer[ns]}
```

Please note the overall system is unnamed. This is our practice in GPAnalyser - each file represents one iGPEPA model, followed by optional Unified Stochastic Probe definitions (or other analyses, which GPAnalyser supports). Due to this, we do not explicitly name system in probes definition either.

B.3 Unified Stochastic Probes syntax

In this section we would like to introduce the syntax for using Probes with GPAnalyser. Again, there are small differences. Below, we present the probes operations quick guide. Please note, that we require all binary operations to be enclosed in brackets for clarity.

$R_l ::=$	$\{pred\}R$	state guarded probe, global probes only
$\mid$	R_l, R_l	sequence
$\mid$	$R_l \mid R_l$	choice
$\mid$	$R_l ; R_l$	both
$\mid$	$R_l : signal$	signal
$\mid$	$R_l[n]$	iterate n times
$\mid$	$R_l[m, n]$	iterate m to n times
$\mid$	$R_l?$	zero or one
$\mid$	R_l+	one or more
$\mid$	R_l*	zero or more
$\mid$	R_l/R_l	reset
$\mid$	$R_l@R_l$	fail
$\mid$	$R_l!$	not
$\mid$	$.$	any action or signal
$\mid$	$action$	eventual specific action or signal
$\mid$	$-\,action$	subsequent specific action or signal
$\mid$	eE	empty action or signal sequence

Predicates, or state guards, stay as originally defined.

For the complete probes, syntax is also very similar to the original. It also keeps the same semantics. Please note that as discussed in the previous section, we do not name the system in the probe definition, which we wish to measure. We simply use the system previously defined in the same file. It will all be easier to demonstrate on examples. We will return to the examples from Subsection 2.5.5. These are represented below:

```
Probe ["output.dat"] (stopTime=250, stepSize=1, density=10)
  steady 500
  {
    GProbe = begin: start, end: stop <-
    observes {LProbe = think: begin, use[2]: end<- }
    where
        {
          Consumers{Consumer[N_c]} =>
                Consumers{(Consumer <think, use> LProbe)
                  | Consumer[N_c - 1]}
        }
  }
```

Informally, the structure is first a Probe definition, its parameters and then global probe specification. Symbols <- specify repetition. Keyword "observes" with the following local probe definition(s) and substition(s) is optional.

The Probe definition is interesting here. First, we specify `Probe` for probing in the fluid flow approximation mode or `SimProbe` for simulation. After that, optional definition of an output file ("output.dat") follows.

Simulation and fluid flow approximation modes differ in the third parameter provided in the parentheses. The first common parameter, `stopTime` specifies the time until which we will be solving/simulating the probed system. The second common argument, `stepSize` is used for increasing the precision of analysis - smaller steps mean higher precision, but take longer to finish. Here, we specified fluid flow approximation mode and the third parameter in such case, `density`, also helps with precision. For `SimProbe`, the third parameter is `replications`, which specifies number of times the simulation will be repeated. For simulation probe, we would therefore start with this definition instead:

```
SimProbe ["output.dat"] (stopTime=250, stepSize=1, replications=5000)
```

Next, the type of probe is chosen. This can be "steady" for steady-state individual passage time, "transient" for transient individual passage time or, the default, global passage time. Both "steady" and "transient" take one numeric argument, which determines the expected steady-state time for the model. In the case of steady-state individual analysis, we will start passage time analysis at this time of model execution. For transient individual passage time, this determines the time for truncation of the unconditional CDF computation, as discussed in Subsection 2.5.4.

For the completeness, we also present the transient individual passage time example from the Subsection 2.5.5, followed by the global passage time example.

```
Probe (stopTime=250, stepSize=1, density=10)
  transient 300
  {
    GProbe = begin: start, end: stop
    observes {LProbe = eE: begin, clear: end }
    where
         {
          Producers{Producer<get_product>Terminal} =>
                Producers{((Producer <get_product> Terminal) <clear> LProbe)
                  | (Producer <get_product> Terminal)[N_p - 1]}
        }
  }

Probe (stopTime=300, stepSize=1, density=10)
  {
    GProbe = eE: start, end[nc * 0.3]: stop
    observes { LProbe = get_product[10] : end }
    where
        {
          Consumers{Consumer[nc]} =>
                Consumers{(Consumer<get_product> LProbe)[nc]}
        }
  }
```

Appendix C

QEST conference tool paper

Below we attached a paper, which we wrote during the course of the project, submitted to QEST 2012 conference. It has already been accepted and will be presented in September, 2012. This paper shortly summarises the implementation contributions of this project and demonstrates the functionality on a simple example.

Specification and efficient computation of passage-time distributions in GPA

Matej Kohut Anton Stefanek Richard A. Hayden Jeremy T. Bradley
Department of Computing, Imperial College London
{mk508,as1005,rh,jb}@doc.ic.ac.uk

***Abstract*—We present a significant extension to the *Grouped PEPA Analyser* tool. We have augmented the tool with the ability to specify complex passage-time distributions with the *Unified Stochastic Probes* formalism and implemented efficient fluid analysis techniques to compute the distributions. The extension incorporates immediate signalling and weighted passive rates and permits two classes of passage time, namely *global* and *individual* passage times, to be computed.**

We summarise how the different classes of passage-time query can be expressed using the Unified Stochastic Probe formalism and present some results from probed GPA models.

I. Introduction

Fluid analysis or mean-field techniques, e.g. [1], allow us to analyse stochastic systems with large populations of identically behaved components. These techniques enable the study of increasingly more complex behaviour described in a variety of formalisms such as process algebras, Petri nets or systems of chemical equations. Traditionally, these give access to the time evolution of means and higher moments of populations of individual component types. However, various derived metrics are often needed, such as the distribution of the time it takes for the system to exhibit a desired observable sequence of actions. In the field of performance analysis, this can be useful when guaranteeing performance or reliability of a system. In particular *Service Level Agreements* (SLA) can be established as a contract between a service supplier and each individual client. For example, an agreement can state that "each client receives a service within $60ms$ at least 99% of the time".

The recent work of Hayden *et al.* [2] describes the *Unified Stochastic Probes* language that allows specification of complex passage-time queries. These can be defined outside of the main behavioural description and then used to produce an extended model. The results from the fluid analysis of this model are transformed to provide passage-time distributions in the original model. This method can quickly check whether the system satisfies a given SLA and can be further used in optimisation frameworks that help with the design of efficient systems.

The probe language allows a concise and user-friendly specification of complex behaviour-based queries. However, the translation to the extended model and the computation of the distribution is too complex to be performed by hand. In this paper, we introduce an extension to the *Grouped PEPA Analyser* (GPA) tool [3] that, for the first time, accepts a sophisticated passage-time query specification, automatically produces extended probed models and then applies fluid analysis techniques to produce the resulting passage-time distributions.

In the following section we describe the syntax added to the tool and the individual improvements to the fluid analysis techniques of GPA that were necessary to fully automate the above method. We illustrate the tool on examples that had to be up until now hand-crafted. We believe that the combination of user-friendly model specification and efficient fluid analysis computation have significant potential for these techniques to be applied in practice.

II. Unified stochastic probes

The Unified Stochastic Probes formalism [2], allows a specification of an observable behaviour of a model. This can be given as a combination of regular expressions that accept sequences of actions exhibited by individual system components as well as logical predicates on the global model state space. Crucially, the resulting *probe* can be attached to the original model without changing the model definition. The fluid techniques then give access to the cumulative density function (CDF) of the distribution of the time it takes for the model to fully complete the specified behaviour.

A. Example

We demonstrate the new features of GPA on a simple producer/consumer model defined in the GPEPA process algebra [4]. The system consists of a large number of producers and consumers. Each consumer can synchronise with a producer to obtain some data. Producers have a buffer that can become full, requiring a reset. The model can be defined in the original GPA syntax:

```
Consumer        = (think, rt).Consumer_get;
 Consumer_get   = (get_product, rg).Consumer_use;
 Consumer_use   = (use, ru).Consumer
Terminal        = (setup, rs).Terminal;
 Terminal_get   = (get_product, T).Terminal
                + (timeout, rti).Terminal;
Producer        = (init, ri).Producer_ready;
 Producer_ready = (produce, rp).Producer_done;
 Producer_done  = (get_roduct, rgp).Producer
                + (clear, rcl).Producer;
Consumers{Consumer[N_c]}<get_product>
  Producers{(Producer <get_product> Terminal)[N_p]}
```

Formally, the system consists of `N_c` consumers and `N_p` producers with attached terminal components, synchronised on the `get_product` action.

In such a system, it would make sense to specify an SLA, for example, one that guarantees that each consumer will not take longer than $250ms$ at least 99% of the time to use two sets of data from the producers. This can be expressed by an individual steady state passage-time calculated by a global probe observing a local probe attached to a single consumer component [2]:

```
Probe ODEs(stopTime=250, stepSize=1, density=10)
  steady {
  GProbe = begin: start, end: stop <-
  observes {LProbe = think: begin, use[2]: end<- }
  where { Consumers{Consumer[N_c]} =>
    Consumers{(Consumer <think, use> LProbe)
      | Consumer[N_c - 1]} } }
```

The specification consists of 4 parts. An analysis is specified (using the standard GPA syntax) that will be used to calculate the passage-time distribution. The keyword `steady` determines that the steady state passage-time will be considered.

The body of the probe first contains a *global probe* definition that determines which pair of signals will be used as the `start` and `stop` of the passage-time. The `observes` block defines *local probes* that will observe individual components and report signals to the global probe. The `where` block defines how the local probes are attached to the system, in terms of a transformation of the model. In this example, we attach the probe to a single consumer component and leave the remaining $N_c - 1$ consumer components unchanged.

To obtain the CDF of the passage-time computed from the first occurrence of the `start` signal instead of the steady state, the keyword `steady` can be replaced by the keyword `transient` and the repeating operators `<-` removed.

Another type of passage-time measure the tool supports is the *global passage time*. This is the time it takes until a *proportion* of given components in a group satisfy a probed behaviour. For example, we can look at the time until 30% of consumers consume 10 products:

```
Probe ODEs(stopTime=300, stepSize=1, density=10) {
  GProbe = eE: start, end[nc * 0.3]: stop
  observes { LProbe = get_product[10] : end }
  where { Consumers{Consumer[N_c]} =>
    Consumers{(Consumer<get_product> LProbe)[N_c]} } }
```

This is achieved by attaching the local probe to each consumer component and stopping the global probe only when the count of `end` signals reaches 30% of N_c. This command gives a point mass approximation to the passage-time when used with the `ODEs` analysis or empirical CDF when used with the `Simulation` analysis respectively.

Each command produces a graphical output of the CDF from each defined global probe and also export the raw data for further processing. Figure 1 shows an example for the three passage-time classes. The transient example is for the passage-time until an individual producer empties its buffer.

B. Implementation details

To support the complete probe syntax and related ODE analyses techniques, following features were introduced to GPA:

- Passive and weighted passive actions [2], to allow more general component specification and for direct translation of probes into GPEPA.

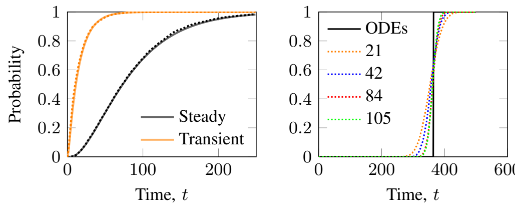

Fig. 1. Plots of individual passage time CDF (left) and global passage-time point mass approximation (right). The dotted lines are obtained from simulation. The global passage-time figure compares the point mass to the CDFs obtained from simulation of the model with increasing scale (the number shown is the value of `N_c`).

- Immediate actions (signals) from iPEPA by implementing *vanishing state removal* [2]. Originally *well-behaved f-components* [2] require only deterministic initial behaviour, but we further restrict this to deterministic signalling paths for all states.
- Full set of Unified Stochastic Probe operations [2] along with the ODE based distribution computation for steady-state individual, transient individual and global passage time shown above. For comparison, these are also implemented in the built-in simulator in GPA.
- Originally, GPA dynamically generates Java classes for numerical solution to the underlying ODE systems. To support more complex models with larger component states, we added the option of dynamic generation of C++ code for the numerical computation.

III. Conclusion & Future work

We introduced an extension to the GPA tool that gives access to complex passage-time measures in large scale systems. The source code is available on the GPA website code.google.com/p/gpanalyser. We tested the techniques on a range of examples, including the large wireless sensor network case study in the original paper [2].

Apart from optimising probe translation algorithms, further improvements include full support for immediate signalling and passage-time computation using higher order moments. We plan to introduce a visual representation of probes and translation from *Performance Trees* [5], making the techniques even more accessible and potentially applicable in practice.

References

[1] J. Hillston, "Fluid flow approximation of PEPA models," in *QEST'05*, pp. 33–42, IEEE, Sept. 2005.

[2] R. A. Hayden, J. T. Bradley, and A. Clark, "Performance Specification and Evaluation with Unified Stochastic Probes and Fluid Analysis," *IEEE Transactions on Software Engineering*, Jan. 2012.

[3] A. Stefanek, R. A. Hayden, and J. T. Bradley, "GPA - A Tool for Fluid Scalability Analysis of Massively Parallel Systems," in *QEST' 11*, pp. 147–148, IEEE, Sept. 2011.

[4] R. A. Hayden and J. T. Bradley, "A fluid analysis framework for a Markovian process algebra," *Theoretical Computer Science*, vol. 411, pp. 2260–2297, May 2010.

[5] T. Suto, J. Bradley, and W. Knottenbelt, "Performance Trees: A New Approach to Quantitative Performance Specification," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pp. 303–313, IEEE, Sept. 2006.

Bibliography

[1] Apache Ant, . URL http://ant.apache.org. 48

[2] ANTLR: ANother Tool for Language Recognition, . URL www.antlr.org. 47

[3] Java Deep-Cloning library. URL http://code.google.com/p/cloning. 47

[4] GCC, the GNU Compiler Collection. URL http://gcc.gnu.org. 48

[5] Guava: Google Core Libraries for Java 1.6+. URL http://code.google.com/p/guava-libraries. 47

[6] Java Development Kit. URL http://www.oracle.com/technetwork/java/javase/overview/index.html. 48

[7] JFreeChart. URL http://www.jfree.org/jfreechart. 47

[8] JUnit. URL http://www.junit.org. 47

[9] Platform Independent Petri net Editor 2. URL http://pipe2.sourceforge.net. 6

[10] Ashok Argent-Katwala, Jeremy T. Bradley, and Nicholas J. Dingle. Expressing Performance Requirements using Regular Expressions to specify Stochastic Probes over Process Algebra Models. In *WOSP'04, 4th International Workshop on Software and Performance*, volume 29 of *ACM SIGSOFT Software Engineering Notes*, pages 49–58, January 2004. URL http://pubs.doc.ic.ac.uk/regular-probes/. 12

[11] Ashok Argent-Katwala, Jeremy T. Bradley, A Clark, and Stephen T. Gilmore. Location-Aware Quality of Service Measurements for Service-Level Agreements. In *TGC'07, Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 222–239, November 2007. URL http://pubs.doc.ic.ac.uk/location-probes/. 12

[12] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying Continuous Time Markov Chains. pages 269–276. Springer, 1996. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3391. 5

[13] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Softw. Eng.*, 29:524–541, June 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1205180. URL http://dl.acm.org/citation.cfm?id=1435631.859038. 7, 24

[14] Christel Baier, Lucia Cloth, and Boudewijn Haverkort. Model Checking Action- and State-Labelled Markov Chains. In *DSN04, Proceedings of International Conference on Dependable Systems and Networks*, pages 701–710. IEEE CS Press, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.1615. 6

[15] Jeremy T. Bradley, Nicholas J. Dingle, Peter G. Harrison, and William J. Knottenbelt. Performance Queries on Semi-Markov Stochastic Petri Nets with an Extended Continuous Stochastic Logic. In *PNPM 2003, 10th International Workshop on Petri Nets and Performance Models, Urbana IL, USA*, pages 62–71, August 2003. URL `http://pubs.doc.ic.ac.uk/ecsl-pnpm2003/`. 6

[16] Darren Brien. Performance Trees: Implementation and Distributed Evaluation. Master's thesis, Imperial College London, 2008. URL `http://pubs.doc.ic.ac.uk/fluid-spa-modelling/`. 6

[17] Federica Ciocchetta and Jane Hillston. Bio-pepa: A framework for the modelling and analysis of biological systems. *Theor. Comput. Sci.*, 410(33-34):3065–3084, August 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2009.02.037. URL `http://dx.doi.org/10.1016/j.tcs.2009.02.037`. 83

[18] Allan Clark and Stephen Gilmore. State-aware performance analysis with extended stochastic probes, 2008. URL `http://www.dcs.ed.ac.uk/home/stg/PEPA/xsp.pdf`. 12

[19] Amer Gerzic. Writing own regular expression parser, November 2003. URL `http://www.codeproject.com/KB/recipes/OwnRegExpressionsParser.aspx`. 22

[20] Marcel C. Guenther and Jeremy T. Bradley. Higher moment analysis of a spatial stochastic process algebra. In *EPEW 2011, 8th European Performance Engineering Workshop*, volume 6977 of *Lecture Notes in Computer Science*, pages 87–101, October 2011. URL `http://pubs.doc.ic.ac.uk/masspa-higher-moments/`. 83

[21] Richard Hayden. Addressing the state space explosion problem for PEPA models through fluid-flow approximation. Master's thesis, Imperial College London, 2007. URL `http://pubs.doc.ic.ac.uk/fluid-spa-modelling/`. 26

[22] Richard Hayden. *Scalable Performance Analysis of Massively Parallel Stochastic Systems*. PhD thesis, Imperial College London, March 2011. URL `http://pubs.doc.ic.ac.uk/hayden-thesis/`. 2, 12, 28, 31, 53, 86

[23] Richard Hayden and Jeremy T. Bradley. A fluid analysis framework for a Markovian process algebra. *Theoretical Computer Science*, 411(22–24):2260–2297, April 2010. URL `http://pubs.doc.ic.ac.uk/fluid-framework-mpa/`. Submitted to TCS, September 2008. Accepted 5 Feb 2010. 26

[24] Richard Hayden, Jeremy T. Bradley, and A Clark. Performance specification and evaluation with Unified Stochastic Probes and fluid analysis. *IEEE Transactions on Software Engineering*, December 2011. URL `http://pubs.doc.ic.ac.uk/fluid-unified-stochastic-probes/`. Submitted 14 August 2010. Revised 24 October 2011. Accepted 24 December 2011. i, 7, 9, 10, 11, 12, 15, 18, 21, 28, 49, 50, 51, 54, 59, 60, 62, 83

[25] Richard Hayden, Anton Stefanek, and Jeremy T. Bradley. Fluid computation of passage time distributions in large Markov models. *Theoretical Computer Science*, 413(1):106–141, January 2012. URL `http://pubs.doc.ic.ac.uk/fluid-passage-time/`. Submitted to TCS November 2010. Accepted July 2011. Available online August 2011. Extended version of June 2009 technical report entitled "Fluid passage-time calculation in large Markov models". 17

[26] Holger Hermanns, Joost pieter Katoen, Joachim Meyer-kayser, and Markus Siegle. Towards Model Checking Stochastic Process Algebra, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.1466. 6

[27] J. Hillston. Fluid Flow Approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2427-3. doi: 10.1109/QEST.2005.12. URL http://www.dcs.ed.ac.uk/pepa/fluidflow.pdf. 24, 25, 26

[28] Jane Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-57189-8. URL www.dcs.ed.ac.uk/pepa/book.pdf. 7

[29] Anton Stefanek, Richard A. Hayden, and Jeremy T. Bradley. GPA - A Tool for Fluid Scalability Analysis of Massively Parallel Systems. In *QEST' 11*, pages 147–148. IEEE, September 2011. ISBN 978-1-4577-0973-9. doi: 10.1109/QEST.2011.26. 2, 25, 39, 45, 59

[30] Tamas Suto, Jeremy T. Bradley, and William J. Knottenbelt. Performance Trees: A New Approach To Quantitative Performance Specification. In *MASCOTS'06, 14th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 303–313, August 2006. URL http://pubs.doc.ic.ac.uk/performance-trees/. i, 2, 6, 31, 39, 86