Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Compositional GSMP simulation with Python

Author:
Miki Ivanovic

Supervisor:
Tony Field

November 1, 2021

Abstract

The Generalized Semi-Markov Process (GSMP) is a framework for modelling and solving discrete event systems. This project proposes an extension to the GSMP for process composition inspired by other compositional process algebras such as Performance Evaluation Process Algebra (PEPA), which brings tractability and computational efficiency benefits over traditional systems.

We develop a flexible Python library for modelling and solving GSMPs, which implements our compositional method. The tool is validated via the simulation of queueing networks, by reproducing the analytic steady-state solution of well-known Markovian systems. This paper mathematically deduces the composition result and verifies its correctness through simulation.

Acknowledgements

I would like to thank my mum, brother and girlfriend for their love and support, which has given me the drive to complete this project.

I also want to extend my acknowledgements to my supervisor Dr. Tony Field for his expertise, patience and guidance. Your commitment to this project has elevated it to a level I did not think possible.

Additionally, I want to thank Dr. Maria Valera-Espina for the all help she has offered me, which has helped get me through some difficult times.

Contents

List of Figures

Chapter 1

Introduction

Discrete event systems are models useful in analysing dynamical problems. Generally, these may be solved either mathematically or numerically by simulation. The Generalized Semi-Markov Process (GSMP) is a formulation for these systems which describes the models through an intuitive automation, forming a simple algorithm for simulation whilst simultaneously setting a mathematical framework capable of analysis.

GSMPs implicitly build a state-space model of systems and solve by generating stochastic traversal paths. When these models can be decomposed into sub-systems, the state space becomes the cartesian product of each corresponding state space, therefore simulation of complex models involves combinatorial calculations that grow exponentially. Our contribution applies techniques from process algebras to form a novel view on composing GSMPs, which allows a single large-scale simulation to be reduced to a sequence of individual, smaller simulations. In turn this avoids the combinatorial issues by inferring global properties of the system via the local properties of each component; thus, improving computational efficiency.

Despite composition having been studied before in GSMPs, this paper takes a different approach by showing that only the event properties are relevant to composition. Therefore, a key challenge is deducing how the system should handle the clocks of events synchronised between multiple processes. We use these results to propose an extension to the framework by defining composition as a sequence of GSMPs synchronising on sets of events. The goal of this project involves developing an efficient, scalable Python environment for solving GSMPs, which implements and verifies our method - then we compare efficiency and tractability between compositional systems and their traditional GSMP equivalents.

In chapter 2 we review the literature on Markov chains, and their relevance to queueing theory – a popular class of discrete event system commonly used in performance modelling. We also summarise composition within the context of PEPA, and the formal definition of the GSMP with its algorithm for path generation; together laying the foundation for the theoretical composition derived in following chapters.

Chapter 3 presents the GSMP simulation environment, we specify the user interface and provide examples of sample programs – demonstrating the simplicity, yet also the benefit of building the tool as a Python library. We dive into the major technical details of the underlying structure within the core.

Composition is introduced in chapter 4; we review the existing literature, then subsequently develop a process-based approach inspired by PEPA. This produces an algorithm, from which the library is promptly extended; we discuss the technical implementation within the core, and again demonstrate the user interface.

Finally, chapter 5 evaluates the accuracy of the tool by simulating queueing networks and reproducing the expected results – providing a proof of work for both the base tool, and the composition extension. We then evaluate the runtime performance of the compositional approach and contrast to a classical GSMP.

Chapter 2

Background

In this chapter we review the definition of a discrete event system, and how they may be solved. We look at Markov chains which can be used to model discrete event system, and outline some powerful results, which are then used to analytically solve certain queueing networks. These are high level formalisms which can be applied to many types of real-world problem.

We delve into process algebras which define a high-level language for reasoning about processes and explore a popular example in PEPA – a formalism for compositional process modelling. Finally, we establish the definition of a GSMP, and how these may be used to simulate a DES.

2.1 Discrete Event Systems

Dynamical systems are mathematical models that evolve over time, and changes are called behaviour. A system is always in some state, thus the behaviour will describe the structure of the states, and the events which influence them. For many dynamic systems, such as network and telecommunications traffic, the behaviours are driven by discrete events – we call these discrete event systems (DES). These events form a discrete random time sequence (over a continuous timeline) indicating when the state changes.

For many systems, while the state is some arbitrary parametrization of the model, we can mathematically define the state space as the set of all possible parametrizations – meaning that at any point in time the system occupies one element of the set. A DES therefore defines the states space, the dynamics of events, and how and when the behaviour occurs. By modelling a real-world problem as a DES, a random sequence of events can be generated to show how the system would behave over time – this is called a sample path, or trajectory.

DES encapsulate a very broad range of models with its generalised definition and have been extensively studied over the past half century; since the DES is presented as a mathematical framework, when certain conditions are imposed, the system may be solved analytically – meaning that properties such as state probabilities or likelihood have succinct formulas. For more complicated systems, many specialised frameworks – or algebras – have been developed for expression and have certain exploitable properties that can lead to numerical, and sometimes analytic, solutions. Finally, simulation – or sample path generation – can be used to obtain numerical solutions.

2.2 Markov Chains

Markov chains are a type of stochastic process (sequence of random variables $\{X_n, n = 0, 1, 2, \}$) that satisfy the Markov property, which states that the conditional distribution of any future X_{n+1} given the previous values $X_0, \cdots, X_n$ will only depend on the value of X_n. This is referred to as the 'memoryless' property in the literature; commonly the values the X_ns take are called states.

The results in this section have been proven by many authors in the literature, here we review some of the elementary results and refer the reader to Harcol-Baker[1] for the proofs (chapters 8 & 12).

2.2.1 Discrete Time Markov Chains

This type of Markov chain is called discrete time (DTMC) because the X_ns form a sequence, the implication being that the system is broken up into discrete time steps so that the sequence records the state at each of these points. Formally:

A DTMC is a stochastic process $\{X_n, n = 0, 1, \cdots\}$ such that $\forall n \geq 0,\ i,\ j$ and $\forall x_0, \cdots x_{n-1}$
$P(X_{n+1} = j | X_n = i, X_{n-1} = x_{n-1}, \cdots, X_0 = x_0) = P(X_{n+1} = j | X_n = i) = P_{i,j}$

The transition probability only depends on the 'current' state; therefore, we have 'stationary' probabilities. These form a *transition matrix* P, where $P_{i,j}$ is the stationary probability of transitioning from state i to j. Clearly $\sum_j P_{i,j} = 1\ \forall i$ to be a valid distribution.

If $P_{i,j} = 0$ then it is not possible to transition from i to j; however, there can exists an intermediate state k such that $P_{i,k} \neq 0$ and $P_{k,j} \neq 0$; so that it is possible to reach j from i within two *jumps*. Extending this idea inductively, two states are said to *communicate* if there exists a sequence of jumps to take the MC from one state to the other, and vice versa. The MC is said to be *irreducible* if all the states communicate with each other.

Going back to the former example, as it is possible to go from i to j in two jumps, what is the probability p of this outcome? Since the jumps are independent, then to take the route $i \ \rightarrow k \ \rightarrow j$, the probability is just the product of the marginal probabilities. Moreover, the overall probability is therefore the sum across *all* possible routes, $p = \sum_k P_{i,k} P_{k,j}$. This defines matrix $P^2 = P \times P$, where the $p = P^2_{i,j}$ and again through induction we can say that $P^n_{i,j}$ gives the probability that in exactly n jumps the MP has gone from state i to j. Mathematically: $\forall n, P(X_n = j | X_0 = i) = P^n_{i,j}$. By taking the limit of the power matrices, $lim_{n\rightarrow\infty} P^n_{i,j} = (lim_{n\rightarrow\infty} P^n)_{i,j}$, the quantity shows the probability of being in state j from i infinitely far into the future.

A DTMC is *ergodic* if it has the following properties:

- *Irreducible*
- *Aperiodic* – the period of a state i is defined as $\gcd n > 0, P(X_n = i | X_0 = i)$, and i is aperiodic if this value is 1. A MC is aperiodic if all its states are aperiodic.
- *Positive recurrent* – the mean time until the MC returns to any previous state is finite. Note that finite-state DTMC are always positive recurrent.

For ergodic DTMCs, there exists a *limiting distribution* $\vec{\pi} = (\pi_0, \cdots)$ where π_i is the limiting probability of being in state i, and that $\pi_j = lim_{n\rightarrow\infty} P^n_{i,j} > 0,\ \forall i$. Moreover, $\vec{\pi}$ is called a *stationary distribution* because it will satisfy $\vec{\pi} P = \vec{\pi}$. This former property yields a convent method for determining the limiting distribution without computing the limit of a matrix power series, specifically the establishing the *stationary equations* $\pi_i = \sum_j \pi_j P_{j,i}$. Moreover, $\pi_i = \pi_i \times 1 = \pi_i \sum_j P_{i,j}$; and linear combination of the two establishes the balance equations:

$$\sum_{j \neq i} \pi_i P_{i,j} = \sum_{j \neq i} \pi_j P_{j,i}$$

By observing that $\pi_i P_{i,j}$ can be interpreted as the jump rate from i to j, the balance equations show that the *total outgoing* rate from i (LHS) is equal to the *total incoming* rate to i (RHS).

Using the balance equations with the identity $\sum_i \pi_i = 1, \vec{\pi}$ (often called the steady state distribution) can be deduced.

2.2.2 Continuous Time Markov Chains

Continuous-time Markov chains (CTMC) extend the DTMC to a continuous timeline, so that the sequence is instead defined as $\{X(t), t > 0\}$ which gives the state at any point in time. The formal definition goes:

A CTMC is a continuous time stochastic process$\{X(t), t > 0\}$ such that $\forall i, j, x(u)$,
$P(X(t+s) = j | X(t) = i, X(u) = x(u), 0 \leq u \leq t) =$
$P(X(t+s) = j | X(t) = i) = P(X(t) = j | X(0) = i) = P_{i,j}(t)$

CTMCs can be converted to DTMCs, meaning that powerful results can be applied directly. Notably the limiting distribution $\pi_j = lim_{t\to\infty} P_{i,j}(t)$ result, which gives the limiting probability of being in state j at time t. When the CTMC is irreducible, the underlying DTMC has ergodicity, therefore the CTMC is called ergodic.

The Markov property makes an important statement about the time spent within each state. Define the holding time Δt_i as the time spent in state i before the CTMC transitions to a new state, then

- $P(\Delta t_i > t + s | \Delta t_i > s) = P(\Delta t_i > t)$
- The probability that the process leaves i in the next t time units is independent to how long it has already stayed in the state.
- The only continuous distribution which has the memoryless property is the exponential

Markov processes (MP) are CTMCs which satisfy this property, and the *embedded* MC is the sequence of states $\{X_n,\ n > 0\}$ which form a sample path of the process, and for each state X_n an associated holding time H_n (exponential random variable) the MP spends in X_n.

A natural way to think about MPs is that they occupy a state i at time t, and that the process awaits an *event* which causes the state transition. The point being that $\forall j \neq i$, the time $T_j \sim \exp(q_{i,j})$ represents how long it will take for the event which transitions from state i to j to occur. The $q_{i,j}$ represents the *rate* of that transition, and a rate of zero is defined to mean the transition is not possible. The events 'race' to trigger first; therefore $H_i = \min_j T_j$ is the minimum of exponential random variables, which itself is exponentially distributed with rate $\sum_j q_{i,j}$.

A question which arises is how to apply the balance equations to a MP, because $P_{i,j}$ are probabilities of the embedded MC. When converting the MP to a DTMC, the balance equations become equivalent to $\pi_i \sum_{j\neq i} q_{i,j} = \sum_{j\neq i} \pi_j q_{j,i}$.

Semi-Markov processes are like CTMCs but may not have exponential holding (also called renewal) distributions, and therefore only satisfy the first Markov property. When the process occupies a state i, the holding time H_i is some generally distributed random variable, which is conditional on the current and next state.

2.3 Queueing Networks

Queueing networks have been used extensively for performance modelling because they provide a graphical abstraction for many stochastic processes with concepts of jobs, delays, and scarce resources; this may range from airport customer flow to network traffic.

Jobs arrive at a queue according to some process; they wait in a buffer before completing and leaving the queue. They are serviced by a server which are resources of the system occupied by the jobs. The order in which jobs are served is known as the queueing discipline. Queues are commonly classified with *Kendall's notation*, that takes the form $A/S/c/K/N/D$:

- A denotes the arrival distribution; M for Markovian/exponential, G for general, and D for deterministic. Characterised by rate parameter λ.
- S is the service distribution and uses the same identifiers as A, but with different rate parameter μ.
- c is the number of servers.
- K is the capacity of the buffer, infinite by default.
- N is the size of the population, also infinite by default.
- D is the queueing discipline; for example, *first-in-first-out* (FIFO), or *processor sharing* (PS) where all jobs are served simultaneously.

There are many other labels, however the default assumption is Markovian with FIFO scheduling. Evaluating performance of a queue is commonly done with the following metrics:

- Response (Sojourn) time T is the total time a job spends within a system, or the difference in completion and arrival time $(T_c - T_a)$.

- Waiting time T^Q is the difference between response and service times, or the time a job spends queueing before service.
- The number of jobs in the queue N^Q.
- The total number of jobs in the system N (jobs in queue plus the jobs being serviced). Often $N(t)$ is used to show the number of jobs at time t, with the limiting value $\lim_{t\to\infty} N(t) = N$.
- Throughput X is the rate of job completion. In a stable system, $X = \lambda$.
- Utilisation ρ is the fraction of time in which the queue is busy.

Queueing networks are directed graphs where the nodes are individual queues called *servers*. The edges show the topology of the system, and the way jobs are routed through the network. Multiple edges may lead out of a node, in which case jobs are *probabilistically routed* to other nodes. Jobs compete for resources, and demand coupled with service rates lead to contention and bottlenecks. There are two classes: **open** and **closed** networks; the former has external arrivals and departures, but the latter does not and therefore have a fixed population N. Moreover, there may be different classes of jobs which traverse the network in different ways and can affect routing probabilities.

One of the most significant results is *Little's Law*, which asserts that for an open ergodic system $E(N) = \lambda E(T)$ and for any closed ergodic system, $N = XE(T)$. This result is extremely useful in computing the former metrics; it may also be applied to T^Q and N^Q.

Many types of Markovian queueing networks have elegant analytical forms, namely they are called *product form* because the nodes operate as independent Markovian queues. Although these networks omit certain interesting features, when a model can be abstracted to such a network then performance measures can be derived easily. The remainder of this section will go into detail on some of these networks and demonstrate these values. The results will be used in later chapters to evaluate accuracy of the software.

2.3.1 M/M/1

M/M/1 represents a single server queue with exponential arrival rate λ, and exponential service rate μ - Fig.2.1. It is the simplest queuing system with a FIFO scheduling discipline. The number of customers in the queue forms a CTMC with transition rate diagram in Fig.2.2.

The limiting state probabilities can be generated from the balance equations:

$$\pi_0 \cdot \lambda = \pi_1 \cdot \mu$$

$$\pi_n \cdot (\lambda + \mu) = \pi_{\{n-1\}} \cdot \lambda + \ \pi_{\{n+1\}} \cdot \mu$$

With solution $\pi_n = \left(\frac{\lambda}{\mu}\right)^n \cdot \pi_0$. This may be verified through back-substitution. π_0 can be determined by taking the sum of probabilities

$$\begin{aligned} &\sum_{n=0}^{\infty} \pi_n = 1 \\ &\Rightarrow \sum_{n=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^n \cdot \pi_0 = 1 \\ &\Rightarrow \pi_0 \cdot \frac{1}{1-\frac{\lambda}{\mu}} = 1 \\ &\Rightarrow \pi_0 = 1 - \frac{\lambda}{\mu} \\ &\Rightarrow \pi_n = (1-\rho)\,\rho^n \end{aligned}$$

Hence forming a geometric distribution with rate parameter $(1-\rho)$, where the utilisation $\rho = \frac{\lambda}{\mu}$. This immediately yields the mean number of jobs in the system:

$$E(N) = \frac{\rho}{1-\rho}$$

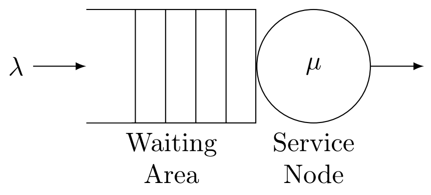

Figure 2.1: Illustration of the M/M/1 queue

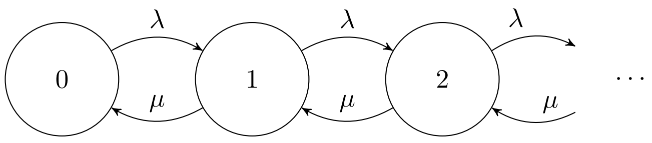

Figure 2.2: The underlying CTMC of an M/M/1 queue

Applying Little's Law yields the mean response time:

$$E(T) = \frac{E(N)}{\lambda} = \frac{1}{\mu - \lambda}$$

Mean queueing times can be obtained as follows:

$$E(R) = E\left(T^Q\right) + \frac{1}{\mu} \Rightarrow E\left(T^Q\right) = \frac{1}{\mu - \lambda} - \frac{1}{\mu} = \frac{\rho}{\mu - \lambda}$$

Finally, applying Little's Law to the mean waiting time will obtain the mean queue size:

$$E\left(N^Q\right) = \lambda E\left(T^Q\right) = \frac{\rho^2}{1 - \rho}$$

Remarkably the M/M/1/PS (processor sharing) also produces the same steady state probability distribution (probability density function - which we refer to as **pdf** from now on). Although the arrival rate is still λ; when in state n there are a total of n customers being serviced all with rate μ/n. However, the time until the next job completes will be the minimum of n exponential distributions with the same rate, implying the rate that state n transitions to state $n-1$ has rate μ – thus producing the CTMC in Fig.2.2.

2.3.2 Jackson Networks

A Jackson network has k servers each with unbounded queues and FIFO discipline; the service times of each server i are exponentially distributed with rate μ_i; and external arrivals into i are Poisson with rate r_i. Jackson networks may have probabilistic routing, where $P_{i,j}$ denotes the probability of customers being routed from i to j. The probability of leaving the system from node i is thus $P_{i,\text{out}} = 1 - \sum_j P_{i,j}$.

Each node has arrival rate $\lambda_i = r_i + \sum_j P_{j,i}\lambda_j$, where $P_{j,i}\lambda_j$ is the throughput from process j. What happens here is that, at equilibrium, a node's throughput is equal to its arrival rate; and that probabilistic routing causes the Poisson arrival streams to split in that ratio. The overall arrival (Poisson) process will be the sum of all the incoming Poisson streams. Therefore, the utilisation of each node is $\rho_i = \frac{\lambda_i}{\mu_i}$.

Jackson networks may be modelled with a CTMC with state vector $n = (n_1, \cdots, n_k)$ such that n_i corresponds to the queue size at node i. Jackson[2] solved the balance equations to obtain steady state probabilities in product form, which establishes the independence of each queueing node in the system.

$$\pi_n = \pi_{n_1,\cdots,n_k} = \prod_{i=1}^{k} \rho_i^{n_i}\left(1 - \rho_i\right)$$

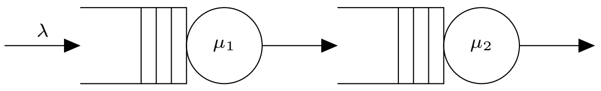

Figure 2.3: Tandem network with two queueing nodes, with service rates μ_1 and μ_2

A tandem queue is an open network comprised of x queueing nodes in series - see Fig.2.3. Customers in queue i will be routed immediately to queue $i+1$, repeatedly until they exit the system. For the scope of this project, we consider the queues to be M/M/1. They are a special type of Jackson network with

$$P_{i,j} = \begin{cases} 1 & \text{if } i+1=j \\ 0 & \text{otherwise} \end{cases}$$

$$r_i = \begin{cases} \lambda & \text{if } i=0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore $\rho_i = \frac{\lambda_i}{\mu_i}$, and the metrics from the previous section can be applied directly to each M/M/1 node.

The BCMP[3] results broadly classify networks with product form solutions, based on the node scheduling discipline. The first result is for networks with FIFO scheduling and unbounded queues; stating that product form exists for open, and closed networks with probabilistic routing given that external arrival rates are Poisson, service times are exponentially distributed, and that the service rate is not class dependent.

Networks may have different types of customers, referred to as classes, which determine the way they are routed through the network and how they are serviced. For brevity we only consider single-class networks.

The BCMP FIFO result is very powerful because it means the nodes may be abstracted as M/M/1 queues, however the last two conditions are restrictive in practice. The second BCMP result for PS networks are stronger, because they still require Poisson arrivals, but the final two conditions are relaxed.

2.3.3 Fork-join

A fork-join subsystem accepts jobs with incoming arrival rates λ, which are divided into equal parts and served in parallel by k FIFO queues. After they have all completed, the tasks are reassembled and then leave the system. The main concern is the systems response time T_k. There is no known general solution for $k \geq 2$; however, an exact form for $k=2$ [4, 5] is $T_2 = \frac{12-\rho}{8}\frac{1}{\mu-\lambda}$. As a result, many of the closed form solutions for more than two servers are computed using approximation and bounding techniques.

We consider fork-join systems where the queues are independent M/M/1, with equal service rate μ; if the external arrivals to the system are Poisson with rate λ, then each queue will have arrival rate λ and utilisation $\rho = \frac{\lambda}{\mu}$.

All the identical parallel queues will have exponential response times with mean $\frac{1}{\mu-\lambda}$. An upper-bound is obtained by assuming the queueing systems are independent, therefore if we call the response time at queue i, $T_k^{(i)}$, and the overall response $T_k \leq \max\{T_k^{(i)}\} = \frac{H_k}{\mu-\lambda}$ is the maximum of k exponential random variables.

Similarly, by neglecting queueing times, the response time at each node equals the service time $\frac{1}{\mu}$, therefore giving the lower bound $T_k \geq \frac{H_k}{\mu}$ where H_k is the kth harmonic number.

Nelson and Tantawi[6] approximate the response time for $2 \leq k \leq 32$ with the formula

$$T_k = \left(\frac{H_k}{H_2} + \frac{4}{11}\left(1 - \frac{H_k}{H_2}\right)\rho\right) T_2$$

2.4 Process Algebra

Process algebras model concurrent processes in a mathematical language, specifically they allow for reasoning on the behaviour and structure of the system. Systems are characterised by active processes (components) and discrete events (actions) that may be taken within them. Actions may represent some internal change within a process, or will be an interaction between multiple processes, thus compositional reasoning is integral to many systems. In pure PAs, the notion of time is abstracted away, so there is no notion of flow, and all actions are instantaneous; however, when a representation of time is desired, a process may be included in the system to 'witness' periods of delay and the actions of each process. Alternatively, some PAs explicitly associate timings to each action.

2.4.1 Performance Evaluation Process Algebra

Performance Evaluation Process Algebra (PEPA) was developed by Hillston [7] in the early 90s for performance measuring. Each action has an associated rate of an exponential distribution that represent delays, forming a tuple which is called activity. PEPA for the following operations:

- $(\alpha,\ r).P$ Basic sequential actions (prefix) within a component. The action α will have a corresponding delay sampled from the exponential distribution with the given parameter r, and becomes process P.
- $P\ +\ Q$ Choice between two or more components, which means that they enter a race to complete an activity – the winner will be the process with the first activity to complete, and the remaining components will discard all their behaviour. This represents a competition between the components for resources.
- $P \bowtie Q$ Cooperation is the compositional operation between processes, which run in parallel and synchronise over a set of shared activities. These activities must be enabled in all their components, otherwise if one of the processes wishes to evolve with a shared action, then it will be blocked until that action can be taken in the rest. Crucially, when an action occurs in a cooperation, the components jointly produce a new activity with a deterministic rate.
- $P \sim L$ Hiding will make a set of activities invisible in a component – the actions are replaced with the generic type τ, but are still allowed to occur in the process with their rate. Hidden activities may be regarded as internal delay.
- Finally, *constants* and *variables* are used for assignment

Note that in each component, there may be multiple instances of the same action type with different rates, that may represent different possible outcomes. When processes cooperate, they have a *synchronisation strategy*, which determines the overall rate of shared activities. There are two common strategies.

Minimum rate can be computed very efficiently, and always chooses the rate leading to the longest delays (slowest). This makes intuitive sense because throughput is always bound by bottlenecks.

Apparent rate of an action of type α in component P, denoted $r_\alpha(P)$, is the sum of *all* the rates of the activities with that type within P – and denotes the rate of a minimum exponential distribution. The apparent rate of all instances of an action within a cooperation is the minimum of the apparent rates within each process – agreeing with the former strategy.

If an instance of action α occurs in P with rate λ, then given an action of type α occurs, the probability it is this activity is $\lambda/r_\alpha(P)$. Assuming process Q has an activity (α, μ), and that P and Q cooperate over a set of shared actions including α, then the probability that these two instances combine to form the shared activity is $\lambda/r_\alpha(P) \times \mu/r_\alpha(Q)$ (assuming independence). As previously stated, in a cooperation when the processes achieve some shared action – the action is interpreted as a new action within the context of the composition, and the model makes a distinction based on the instances chosen. The activity corresponding to the occurrence of (α, λ) in P and (α, μ) in Q is $(\alpha,\ R)$ in $P \bowtie Q$, where $R = \lambda/r_\alpha(P) \times \mu/r_\alpha(Q) \min(r_\alpha(P), r_\alpha(Q))$.

Bradley *et al.*[8] demonstrate that the use of minimum rate strategy leads to marginal errors in performance metrics; however, the practical user benefits present a trade-off for the accuracy. Specifically, whilst apparent rates will produce precise minimum rates at the system level; the computational overhead is too expensive for simulation – and that a clever design approach is necessary for a practical implementation.

2.5 Generalized Semi-Markov Process

The generalised semi-Markov process (GSMP) is a framework for stochastic discrete-event systems which generates a state space S through the stochastic timed automation

$$(S, E, E(s), p, F, r, S_0, F_0)$$

- S is the countable state space.
- E is the countable event set.
- $E(s)$ is the set of events scheduled to occur in state s.
- $p(s', e, s)$ is the probability of state transition $s \to s'$ via event e.
- $F(x; s', e', s, e)$ is the distribution function with parameter x setting the 'clock' of e' after transition $s \to s'$.
- $r(s, e)$ is the clock rate of event e in state s.
- S_0, F_0 are the distribution functions yielding the initial state and event clocks respectively.

The DES occupies a unique state s at any point in time, and active events $E(s)$ compete to fire. Events have an associated clock random variable C_e which reads a time-to-live (residual lifetime) before the event triggers. The clock reading for event e will decrease at rate $r_{s,e}$ – therefore the time-to-live can depend on the physical state occupied by the DES. Ultimately the next transition will be the one the event e^* that triggers first.

Observe that here all these properties are conditional on the immediate history of getting to the current state, i.e., the previous state and winning event. This is a specific class of GSMP called **time homogeneous**, and the sequence of state form an embedded MC. **Time inhomogeneous** GSMPs will have probabilities, and lifetimes conditional on the *entire* history of the process – capturing the general behaviour of *any* DES; however, this leads to high computational costs. Moreover, performance modelling generally makes assumptions and simplifications of systems, thus making it easy to reason about behaviour. For this reason, the software discussed in the following chapters implicitly models this class of GMSP, but still offer ways to *track* history and effectively produce time inhomogeneous processes.

Glynn [9] demonstrated it is possible to reason about GSMPs, and that theoretical results can be established – presenting the main motivation for their use.

2.5.1 Sample path generation

A new physical state is randomly chosen, depending on both the previous state and the trigger event by $p(s', e^*, s)$. Old events $(O(s'; s, e^*) = E(s') \cap (E(s) - e^*))$ continue to tick down, and new events $(N(s'; s, e^*) = E(s') - O(s'; s, e^*))$ will need to have their clocks scheduled by F.

This procedure, outlined with the following pseudocode, can iteratively generate a path through the state space.

```
1. Use $S_0$ to select the initial state $s$, and use $F_0$ to initialise all active event clocks in this state.
2. Loop:
3. Let $t_{\text{next}} = \min\left(\frac{C_e}{r(s,e)}, e \in E(s)\right)$ and $e$ be the winning event.
4. Determine the next state $s'$ using $p(.; s, e)$.
5. For each old event $o \in E(s')$ reduce $C_o$ by $t_{\text{next}} \times r(s, o)$.
6. For each new event $n \in E(s')$ set $C_n = F(.; s', n, s, e)$.
```

7. Set $s = s'$.
8. End loop

This procedure nominally extends the concepts introduced in Sec.2.2.2; the GSMP is a renewal process such that the current state holding time is the minimum of a set of random variables, conditional not just on the current state and the time within it, but also on the previous state and trigger event (when we consider new events). Of course, the distributions may be defined in such a way that they are not conditional on these parameters; specifically, if the residual lifetimes of each event are always exponential – then the GSMP models a MP. Furthermore, if the residual lifetimes are generally distributed and conditioned on the current and previous states, then it is a SMP. In both these cases the model does not need to use the full scope of available history, analysis of these processes is (generally) simpler than that of some arbitrary GSMP.

Chapter 3

GSMP Python Library

Here we give the GSMP simulation environment, we first outline the design requirements, and then present an overview of the user interface - namely on how to construct models and then solve. We then discuss the technical implementation within the core, which transforms the GSMP specifications to a tangible algorithm for simulation. Finally we present some optimisations which improve the library's efficiency.

3.1 Design objectives

The main objective of the library is to offer a simple framework for GSMP simulation. The software must be reliable by guaranteeing accuracy and efficiency. Furthermore, it should be easy to integrate with third party tools and other programs – and should ultimately afford users a natural approach to modelling GSMPs.

3.2 Overview of GSMP simulation

The library consists of a core package which provides the classes `GSMP`, which is used to define models, and `Simulator` used to generate sample paths.

3.2.1 User interface

The GSMP abstract class acts as an interface for defining models, the user creates a custom object that inherits from this superclass – and will then write several functions corresponding to the automation from Sec.2.5. The interface outlines the argument patterns and expected return types; however, it is at the discretion of the user to ensure correct typing because the duck typing of Python will not throw any errors until runtime. States and events can be defined as any immutable type, such as strings or numbers.

The interface defines the following instance methods, which directly correlate to the GSMP automation:

- `states(self)` returns an "iterable", python's class of iteration data structures (such as lists, dictionaries, sets, generators, . . .), of states.
- `events(self)` returns an iterable of events.
- e(self, s) returns an iterable of events, which are active in s.
- `p(self, s', e, s)` returns a number in the range [0, 1]; and defines a pdf over the event set e(s) to the adjacent states s'.
- `f(self, s', e, s, e)` returns a non-negative float, which is sampled from some distribution.
- `r(self, s, e)` returns a positive float.
- `s_0(self, s)` returns a number in the range [0, 1]; defines pdf over the state space.

- `f_0(self, s, e)` returns a non-negative float, sampled from a distribution.

Note that `self` is the python instance parameter. It is not necessary to define a constructor, however if done so then a call must be made to the super class. The default constructor has an optional keyword argument which is discussed in Sec.3.4. GSMP objects may be composted via the addition operator, see Ch.4.

The `Simulator` class is constructed with a GSMP object, and may call the function `run()`. On each run, the GSMP process have their internal states and clocks reset and proceed to generate sample paths in a loop. Run accepts the following keyword arguments:

- `epochs`: the positive integer number of path traversals to make, infinite by default.
- `warmup_epochs`: warmup path traversals generate before the actual run to help reach system stability, zero by default
- `until`: positive time float specifying the maximum internal runtime of the simulation, infinite by default
- `warmup_until`: positive float specifying a maximum internal runtime for a warmup run, zero by default
- `estimate_probability`: boolean flag indicating return type, false by default
- `plugin`: function accepting a single argument, None by default.

Note that if both `warmup_epochs`, and `warmup_until` are provided, then two separate warmup runs are conducted. Conversely, if both `epochs` and `until` are supplied, the simulation continues until the first one is reached.

An example program is supplied in Fig.3.1, which models a simple M/M/1/k queue.

3.2.2 Program output

Any results processing is handled by the user via the `plugin` argument. On each simulation cycle, the plugin function is called with the dictionary object containing the following keys:

- `"old state"`: old state
- `"new state"`: new state
- `"event"`: trigger event
- `"process"`: trigger event parent process
- `"time"`: cycle time

`run()` has the `estimate_probability` flag which returns a list of tuples, of observed states with their observed probability (obtained by dividing the total state holding time by the total run time); however it is disabled by default for performance reasons. To obtain more meaningful performance metrics, the simulation will stream transition data to the plugin function and leaves parsing at the programmer's discretion. An sample plugin is provided in Fig.3.2, which shows how response and waiting times from a M/M/1/(k) can be extracted from the simulation data. Fig.3.3 shows how this plugin and the M/M/1/k program can be simulated.

3.3 Core implementation

Simulator implements the path generation algorithm outlined in Sec.2.5.1 by breaking up the various stages into different functions and coordinating them in `run()`. It first applies the pre-processing step 1, then loops over the path generation steps 3-7 several times depending on the input parameters (such as warmup).

Note that the automation defines the model at a high level of abstraction, for this reason the hierarchical structure in Fig.3.4 is defined for GSMP models, which converts definitions into tangible methods.

```python
from core import Gsmp
from numpy.random import exponential

k, arrival_rate, service_rate = 20, 1, 2


class MM1k(Gsmp):
    def states(self):
        return range(self._k + 1)

    def events(self):
        return ["arr", "com"]

    def e(self, s):
        es = []
        if s < self._k:
            es.append("arr")
        if s > 0:
            es.append("com")
        return es

    def p(self, _s, e, s):
        if e == "arr":
            return int(s + 1 == _s)
        else:
            return int(s - 1 == _s)

    def f(self, _s, _e, s, e):
        if _e == "arr":
            return exponential(self._avg_arrival_time)
        else:
            return exponential(self._avg_service_time)

    def r(self, s, e):
        return 1

    def s_0(self, s):
        return int(s == 0)

    def f_0(self, s, e):
        return exponential(self._avg_arrival_time)

    def __repr__(self):
        if self.name is None:
            return hex(id(self))
        return str(self.name)

    def __init__(self, name=None, k=k,
                 _arrival_rate=arrival_rate, _service_rate=service_rate):
        self.name = name
        self._k = k
        self._avg_arrival_time = 1 / _arrival_rate
        self._avg_service_time = 1 / _service_rate
        super().__init__()
```

Figure 3.1: M/M/1/k GSMP model in Python. `NumPy`'s exponential distribution is used in `f`.

```python
from collections import deque

arrivals = deque()
response = deque()
waiting = deque()
service_times = deque()


def plugin(data):
    t = data["time"]
    e = data["event"]

    if e == "arr":
        arrivals.append(t)
        if data["old state"] == 0:
            service_times.append(t)
    else:
        response.append(t - arrivals.popleft())
        waiting.append(response[-1] - (t - service_times.popleft()))
        if data["new state"] > 0:
            service_times.append(t)
```

Figure 3.2: M/M/1 simulations will return a time-sequence of events, by using some lists we can determine the service time, response time and waiting time of each job with a simple algorithm.

```python
from core import Simulator
from mm1k import MM1k
from mm1_plugin import plugin, response, waiting

if __name__ == "__main__":
    res = Simulator(MM1k()).run(
        until=1000,
        estimate_probability=True,
        plugin=plugin
    )
    print(res, response, waiting)
```

Figure 3.3: We simulate the M/M/1/k model from Fig.3.1 using the plugin from Fig.3.2, and print the response and waiting times, and the observed pdf.

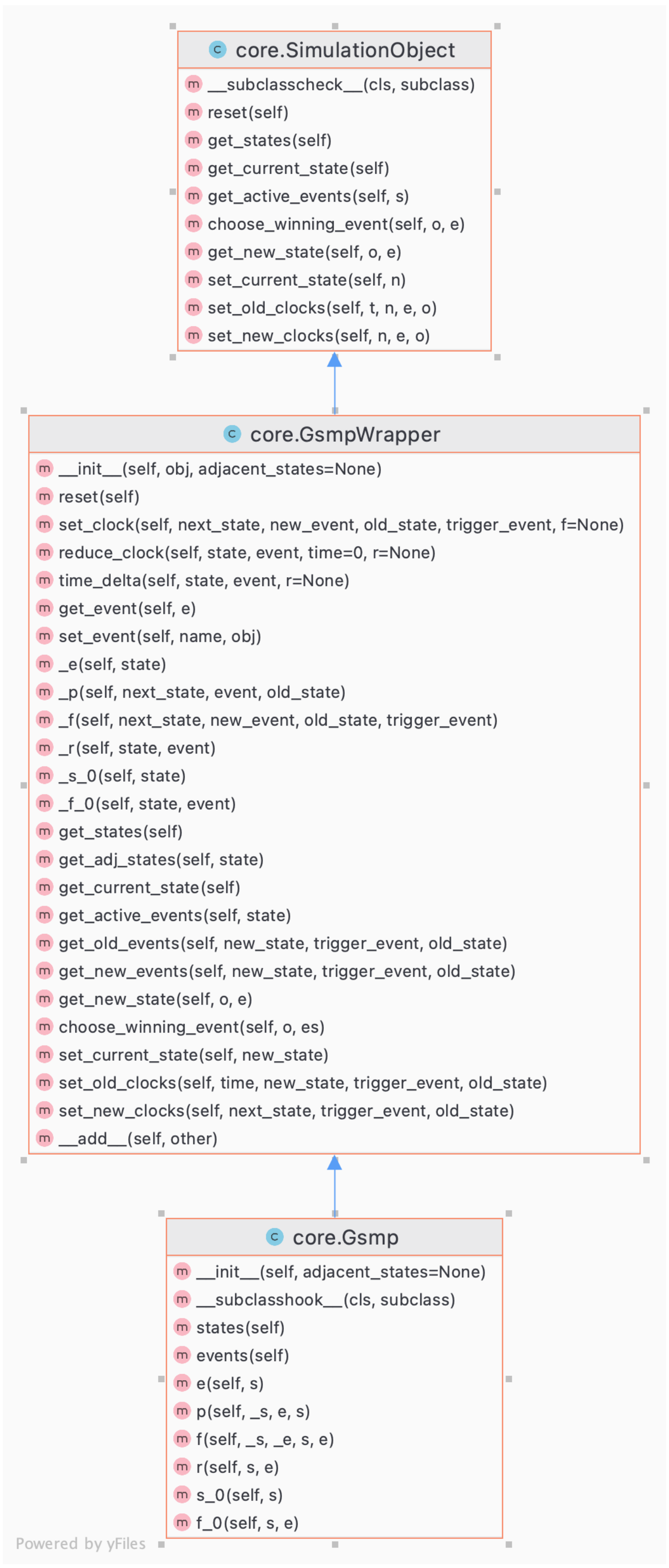
core.SimulationObject
__subclasscheck__(cls, subclass)
reset(self)
get_states(self)
get_current_state(self)
get_active_events(self, s)
choose_winning_event(self, o, e)
get_new_state(self, o, e)
set_current_state(self, n)
set_old_clocks(self, t, n, e, o)
set_new_clocks(self, n, e, o)
core.GsmpWrapper
__init__(self, obj, adjacent_states=None)
reset(self)
set_clock(self, next_state, new_event, old_state, trigger_event, f=None)
reduce_clock(self, state, event, time=0, r=None)
time_delta(self, state, event, r=None)
get_event(self, e)
set_event(self, name, obj)
_e(self, state)
_p(self, next_state, event, old_state)
_f(self, next_state, new_event, old_state, trigger_event)
_r(self, state, event)
_s_0(self, state)
_f_0(self, state, event)
get_states(self)
get_adj_states(self, state)
get_current_state(self)
get_active_events(self, state)
get_old_events(self, new_state, trigger_event, old_state)
get_new_events(self, new_state, trigger_event, old_state)
get_new_state(self, o, e)
choose_winning_event(self, o, es)
set_current_state(self, new_state)
set_old_clocks(self, time, new_state, trigger_event, old_state)
set_new_clocks(self, next_state, trigger_event, old_state)
__add__(self, other)
core.Gsmp
__init__(self, adjacent_states=None)
__subclasshook__(cls, subclass)
states(self)
events(self)
e(self, s)
p(self, _s, e, s)
f(self, _s, _e, s, e)
r(self, s, e)
s_0(self, s)
f_0(self, s, e)
Powered by yFiles

Figure 3.4: UML of the core hierarchy. The proxy class `GsmpWrapper` translates `Gsmp` models into `SimulationObjects`.

The top level is the interface `SimulationObject`, which defines the instance methods used for simulation – and is the actual type of the model passed to the `Simulator` class. `Simulator` applies the `SimulationObject` methods to achieve high-level simulation.

At the bottom, we have the `GSMP` interface, which programmers use to define their models. But how does the `core` convert these functions to the necessary ones for `Simulator` to work?

The `GSMP` interface inherits from a proxy class which tracks the current state, and has a dynamic HashMap from events to their clock values; it implements the `SimulationObject` interface methods:

- `reset(self)`: selects the first state and initialises active event clocks using S_0 and F_0 respectively.
- `get_states(self)`: returns an iterable of the state space.
- `get_active_events(self, state)`: returns a list of active events in `state`.
- `get_old_events(self, new_state, trigger_event, old_state)`: returns O(`new_state`, `trigger_event`, `old_state`).
- `get_new_events(self, new_state, trigger_event, old_state)`: returns N(`new_state`, `trigger_event`, `old_state`).
- `choose_winning_event(self, o, es)`: From the set of active events `es` in old state `e`, returns a tuple `(event, time)` of the winning event and its remaining lifetime in state `o`.
- `set_old_clocks(self, time, new_state, trigger_event, old_state)`: decrements all the old clocks in O(`new_state`, `trigger_event`, `old_state`) by `time` $\times$ `r(s, e)`.
- `set_new_clocks(self, next_state, trigger_event, old_state)`: sets all the new clocks in N(`next_state`, `trigger_event`, `old_state`) with the GSMP `f` function.
- `set_current_state(self, new_state)`: updates the state tracker.

The proxy class also implements the private `adjacent_states` function, which may be overridden via the constructor, therefore the GSMP interface has a call within its constructor to the superclass. The wrapper has two more features, it raises errors if a GSMP definitions are missing; and defines the addition operator as composition, so when two GSMP instances are added together with the infix operator '+', a compositional GSMP is implicitly created – see Ch.4.

3.4 Optimisations

Memoization trades expensive function calls for memory space, by caching results on the first call, performing fast lookups from then on. Both the active event sets and the adjacency lists - Sec.3.4.1 - are static properties of states. We achieve this by wrapping the `@cache` decorator from the built in library `functools` over instance methods in the `GsmpWrapper` class.

3.4.1 Adjacency lists

When the algorithm probabilistically determines the next state, it only needs to consider the 'adjacent' states that can be reached with a single transition. These might be obvious, for example the adjacent states in a simple queue will just be state± 1 as each event moves only one customer.

However, the machine doesn't understand the semantics of the model; therefore, when choosing a new state after the winning event has fired, it will need to calculate probabilities and make a weighted choice across the entire state space. In other words, an $O(|S|)$ overhead at each jump. This can be mitigated by having mapping function from each state to its neighbours, so that probabilities are only computed across the relevant state subset.

Unfortunately, the cost of computing adjacent states $\{s' | p(s', e^*, s) > 0, e^* \in E(s), s' \in S\}$ is high because it requires a full state-space traversal per active event, with an $O(|S| \times |E(s)|)$ overhead. When the GSMP calculates state transition probabilities, a full state-space traversal is needed anyway, so by pruning to only the relevant states many redundant calls to the probability function can be avoided. Moreover, picking a winning state from a smaller weighted set will be computationally efficient.

This is the worst-case scenario, but as previously stated adjacent states can be easy to deduce within context – so the GSMP class takes an additional (optional) parameter accepting custom adjacent functions (like $E(s)$) to override the default. With this, the overhead of obtaining the adjacency list can be reduced to $O(1)$ per iteration. Furthermore, expensive traversals of the state space are avoided through this approach, resulting in lazy evaluation. There is an added benefit in that infinite state space data structure become supported with a clever implementation.

The keyword argument `adjacent_states` in a GSMP constructor accepts a function which takes a single state parameter. If it is not given, then the system will default to the inefficient brute force solution on every cycle. For an example, we refer the reader to Appendix.A.1,A.2.

Chapter 4

Composition

We motivate the use of composition for GSMPs, and review existing methods within the literature. We then present a novel approach to compositional GSMPs by analysing how such a model would be solved via simulation. After deducing the results, we demonstrate how to build a compositional GSMP with a sequence of sub-GSMPs that synchronise over shared events. This idea is then implemented within the Python framework, and we discuss how to translate to code; finally we give a brief description of the technical implementation.

4.1 Motivation

The GSMP state vector holds information about various internal properties of systems. As model rise in complexity and size – combinatorial explosions lead to intractability. Consider the earlier examples of M/M/1 queues and the tandem queue from Sec.2.3.

The former will have a state space equal in length to the natural numbers, and two events denoting arrival and service at the queue – therefore implementation and modelling is straightforward and simple. With the latter being a composition of n such queues, the state space has become the Cartesian product of n sets of natural numbers - see Fig.4.1. Some events become *synchronised* such as arrivals and service between adjacent queues – however the duplicates can be abstracted away in the context of the model. Nevertheless, the size of the event set is proportional to the number of tandem queues; but clearly the state space grows exponentially causing high computational overhead.

Although the tandem queue might still be relatively easy to reason about because the edge cases arising from the composition of two tandem queues can be extended to arbitrary n – it still demonstrates the tractability issues. Another example might be the M/M/c queue, which extends the M/M/1 ideas by allowing for an arbitrary number of servers. The state of such a queue would need to be a $c + 1$ dimensional vector, as the internal state would comprise of the queue length and status of each server.

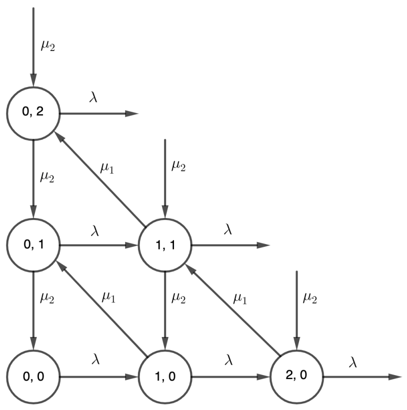

Figure 4.1: CTMC of two-M/M/1-queue Tandem system.

Many systems can be thought of as a composition of various sub-processes which synchronise on sets of shared events; process algebras are used to reason about behaviours, and this often easier than analysing a large GSMP. This is an example of divide and conquer – but the framework in Sec.2.5.1 has no notion of composition or parallelism, therefore many larger models become syntactically bloated. Moreover, the explicit computation of combinatorial state spaces makes simulating larger systems infeasible; however a compositional approach in which the state space is never explicitly computed, but rather inferred from the internal states of each process - would theoretically avoid these issues.

The objectives of this chapter involve developing composition ideas for GSMPs: by taking a sequence of small GSMPs which synchronise over shared events, we can solve the tractability and computational issues with the GSMP. We first assess the literature on GSMP composition, and then deduce the theoretical properties of path generation. Finally, by using these novel ideas, an extension to the framework is proposed, and we then discuss the practical implementation within the Python GSMP library.

4.2 Process separability

Nilsen's[10] approach to composition establishes a *separability* condition on GSMP components. The main idea is to take a composition of processes, and to introduce proxies to change the conditional structure of the components. Consider two components X and Y; each process will have an internal embedded MC, thus in general the probability distributions will be conditional on both X and Y. But, if the synchronised event flow was from X to Y, then the pdf of X is conditioned only on itself.

The proxy component X' is added as a buffer between the two; having an internal state and event set of its own – however these are inferred from the set of shared events. X' also has its own pdf, and clock setting functions which are conditioned solely on X. It also has old and new events sets conditioned similarly. All the corresponding properties of Y become conditioned on both X' and itself; therefore X and Y have become independent.

If the systems were to allow for two-way event flow between two components, then two separate buffers would need to be added to the system to channel directional flow. The benefit of this approach is that it scales to complex models, and in GMSim they are abstracted as interfaces. Nevertheless, it differs from the compositional approach in PEPA because there is no concept of blocking and apparent rates. The tacit assumption that events flow from a *source* process enables it to evolve independently, but the proxy component can add delays to the evolution of the *receiving* process; the state of the buffer must be included as a component of the global state to justify any performance consequences. Hence imposing restrictions on the type of system which may be modelled.

4.3 GSMP path generation in a compositional context

Define process $X = (S, E, E(\cdot), p, F, r, S_0, F_0)$ as the composition of the well-defined sub-processes $\{X_1, \cdots, X_n\}$ where $X_i = (S_i, E_i, E_i(\cdot), p_i, F_i, r_i, S_{0,i}, F_{0,i})$. Then clearly $S \subseteq \times S_i$ and $E = \cup E_i$. We aim to deduce the automation of X given those of its sub-systems.

A composition of processes needs to synchronise over events, therefore the complexity will come from event-related procedures rather than state. There are two general cases to consider.

4.3.1 Composing independent GSMPs

Suppose each E_i is disjoint, meaning that there are no shared actions between any sub-process, implying each X_i is *independent*. Consider X occupies $s = (s_1, ...s_n)$, then a state transition goes as follows:

- Each X_i occupies s_i.
- $E(s) = \cup E_i(s_i)$ gives the active events since there is no blocking.
- Each $e \in E(s)$ will 'tick down' at a rate solely determined by its own process (due to independence).

- Let event $e^* \in E_i$ be the winner, therefore the inter event time $\Delta t = \frac{C_{e^*}}{r_i(s_i, e^*)}$.
- Let the next state be $s' = (s'_1, \cdots, s'_n)$; because e^* belongs exclusively to X_i, no state transitions beyond the scope of e^* will occur, thus $s_j = s'_j \forall j \neq i$.
- Therefore to deduce s', only s'_i needs to be chosen from $p_i(s'_j, e^*, s_i)$.
- Within all $X_j, j \neq i$, the active events have all become old and need to be updated to reflect the passage of time. Each $o \in O_j(s'_j, s_j, e^*) = E_j(s_j)$ are decremented by $\Delta t \times r_j(s_j, o)$ since the clock speeds are exclusively determined by the process X_j.
- In X_i, the old clocks are updated normally.
- The only new clocks in X must be the new events in X_i; so $N(s', s, e^*s) = N_i(s'_i, s_i, e^*)$. These events e can be set by the internal $F_i(\cdot, s'_i, e, s_i, e^*)$.

The key observation is that independence event clocks can be manipulated within the unique parent sub-process, resulting in behaviour which nicely extends the original path traversal. In effect, each process is allowed to evolve freely.

4.3.2 Synchronising events

Now introduce event e' over which X_1 and X_2 synchronise, so that the shared event will trigger changes to the internal states of both respective processes when fired. The two are no longer independent. Consider the same scenario outlined at the start of the last section. Note that here we are abstracting any event names, so the unique label e' is used to denote the two events from X_1 and X_2 which are being synchronised - though in practice they might have different names. Moreover synchronisation implies that shared events always fire at the same time, therefore e' only needs a single clock - we can discard both the corresponding clocks and events in the individual components and replace with this global instance.

When is e' active in X, given the current state $s = (s_1, s_2, s_3, \cdots)$? Clearly if $e' \in E_1(s_1)$ and $e' \in E_2(s_2)$, e' is active in both X_1 and X_2 therefore it is active in X. Likewise, if e' is not active in either processes then it cannot be active in X. However what happens when it is active in only one?

Without loss of generality, let $e' \in E_1(s_1)$ but $e' \notin E_2(s_2)$; if we were modelling this system in PEPA, when X_1 tries to evolve with e', it will be blocked until X_2 can also fire the event. Thus we deduce that $e' \notin E(s)$, however this implies that although e' will never fire in s, there *may* be other events in both X_1 and X_2 which *can* still trigger - so neither process gets blocked by X. This is where the model differs from an operational system - instead of blocking the *process* we block the *event* from firing in X_1.

Suppose that $e' \in E(s)$, then the time until e' fires is $C_{e'}/r(s, e')$. The clock speed is conditional on both X_1 and X_2, so neither r_1 nor r_2 are sufficient towards determining the true rate because they are defined only within the scope of X_1 and X_2 respectively. Due to the generalised nature of the sub-processes there is no way to determine r directly from r_1 and r_2; therefore r will be a property of the composition.

Let e' win with time Δt, and the next state be $s' = (s'_1, s'_2, s'_3, \cdots)$. Clearly $s'_j = s_j \forall j \neq 1, 2$ because the only state changes occur within the sharing processes of e'. Furthermore, despite e' being shared by X_1 and X_2, the way it manipulates their internal states is still independent event though the processes are not; so $p_1(s'_1, e', s_1)$ and $p_2(s'_2, e', s_2)$ are used to choose s'_1 and s'_2 respectively.

Synchronised old events $O(s', s, e')$ can be computed similar to $E(\cdot)$, i.e., an event is old if and only if it is old within all its processes. To justify the claim, let $\hat{e} \neq e'$ be shared between X_1 and $X_3 \in X$, and consider the following cases:

- If $\hat{e} \notin E(s)$, then $\hat{e}$ is not an old event, and cannot be old after the transition e'.
- If $\hat{e} \in E(s)$, then $\hat{e}$ lost the race to e' (the trigger event) and it will clearly be old in X_3. Furthermore, if $\hat{e}$ is now old in X_1, then it is clearly old in X; otherwise it has been cancelled and is therefore blocked in X_3.

As mentioned above, the clocks of shared old events must be decremented based on the global rate; however all other events can be handled in their unique process.

New events $N(s', s, e')$, like above, will solely have events from $N_1(s'_1, s_1, e')$ and $N_2(s'_2, s_2, e')$. Any non-shared event (call these events *unique*) from $N_1 \cup N_2$ are active in s' and are new; however the shared events are only valid if they are active in the new state - meaning active in all of their parent processes. This follows from the definition of new events $N(s', s, e^*) = E(s') - O(s', s, e^*)$, and implies that once shared events are unblocked their clocks are reset. Unique new events can have their clock set in their parent process like before, but for shared new events - much like rates - the individual clock setting functions are generally insufficient because the true F is a conditional distribution.

We remark that when the system has a unique trigger event, or any of the old/new events are non-synchronised, the process may be reduced to the independence case in Sec.4.3.1.

These notions can be inductively extended to compositions of many sub-processes with many shared events. By combining the ideas presented in this section, a formal definition of compositional process X can be determined from its components $\{X_i\}$ as follows:

- $S \subseteq \times S_i$; $E = \cup E_i$.
- $E(s) = \{e | e \in E_i \Rightarrow e \in E_i(s_i), \forall i\}$.
- $p(s', e, s) = \prod p_i(s'_i, e, s_i)$.
- $F(\cdot; s', e', s, e) = F_i(\cdot; s'_i, e', s_i, e)$ if e' and e are both unique to X_i.
- $r(s, e) = r_i(s_i, e)$ if e is unique to X_i.
- $S_0(s) = \prod S_{0,i}(s_i)$.
- $F_0(\cdot; s, e) = F_{0,i}(\cdot, s_i, e)$ if e is unique to X_i.

Moreover, $O(s', s, e) = \{o | o \in E_i \Rightarrow o \in O_i(s'_i, s_i, e), \forall i\}$; and $N(s', s, e) = \{n | e \in E_i \Rightarrow n \in N_i(s'_i, s_i, e) \wedge n \in E(s'), \forall i\}$. Note that definitions for F, r, and F_0 are given only when they are provided with unique events. In the Independence case discussed in the above section, these definitions are sufficient; however, additional definitions (overloading) for these clock functions must be provided for non-unique events.

In Sec.2.4.1 we discussed synchronisation strategies for cooperation, and that in real world examples components may disagree on the clock setting functions for shared events. Those strategies would be implemented in F, r, and F_0; however notice that in the case of PEPA, all actions are memoryless, hence the justification of the apparent rate strategy. Compositional GSMPs inherently offer more flexibility because the event distributions are general; therefore it is not guaranteed that components will define event clocks from the *same* distribution, let alone with the same rate parameters. Thus we do not explicitly define any such strategies here; nevertheless, this model implicitly follows the cooperation of PEPA.

To summarise, we can define GSMP composition $X = X_1 + X_2$ for two complete GSMPs X_1, X_2, over the shared event set E by specifying functions F, r, F_0 (defined solely over E).

4.4 Compositional modelling in Python

In this section we extend the library from Ch.3 to support compositional GSMP from the Sec.4.3. The goal is to use these in developing process driven composition, namely by building a new `SimulationObject` that may be used in the same way as normal GSMP objects. The composition operation needs to be minimal and streamlined, akin to a process algebra – yet more powerful because it will allow the model reuse. The general strategy within the core will be to use object-oriented paradigms and to defer as much logic to the underlying component GSMPs. We effectively simulate the components in parallel, and allow to composition to make choices on clock setting and event selection based on results from the previous section.

```
from core import Compose
from queues import queue1, queue2, queue3

from shared_events import shared_events

from clock_functions import f, r, f_0

tandem_queue = Compose(
    queue1, queue2, queue3,
    shared_events=shared_events,
    f=f, r=r, f_0=f_0
)
```

Figure 4.2: Composition of three queue M/M/1 tandem network with `Compose()`.

```
from queues import queue1, queue2, queue3

from shared_events import shared_events

from clock_functions import f, r, f_0

tandem_queue = queue1 + queue2 + queue3
tandem_queue.shared_events = shared_events
tandem_queue.f = f
tandem_queue.r = r
tandem_queue.f_0 = f_0

if __name__ == "__main__":
    from core import Simulator
    Simulator(tandem_queue).run(until=1000)
```

Figure 4.3: Composition of three queue M/M/1 tandem network with '+' operator. Note that for both this figure, and Fig.4.2, the `shared_events` are defined in Fig.4.4, and the clock functions in Fig.4.5.

4.4.1 Framework and usage

The `core` has an additional `Compose` class for modelling compositional GSMPs. The desired GSMP models are given as arguments to the constructor - see Fig.4.2, the ordering of which determines the state vector, for example the first model in the sequence will occupy the first position. As discussed in the Sec.3.3, composition may be achieved with the use of the infix '+' operator (Fig.4.3, which internally constructs `Compose` objects. This means that functional reduce operations, or comprehensions can be used over this operator to efficiently construct large systems.

Synchronised events may be either passed into the construction via a `shared_events` keyword argument or set directly with an assignment to the objects `shared_events` property after initialisation. Custom implementations of clock functions `f`, `r`, and `f_0` may be similarly assigned.

4.4.2 Shared events

Events have two identifying properties – the corresponding *parent* process, and a unique immutable `name` within the process. Unfortunately, the condition is not that the name is unique within the global scope, rather only within the parent's, therefore, event equality depends on `both` values in a composition.

When the programmer seeks to define the set of synchronised events, the information must derive this equality. For this a tuple format is adopted – `(name, process)` – the process corresponds to the instance of the GSMP model(see Fig.4.4). A shared event is denoted by an iterable data structure of these tuples – with the condition that each event belongs to a unique process. Once this mapping has been defined, the first item as the **default** event identifier (we call corresponding process *default*). The default event will be the tuple that is returned to the programmer on

```
from queues import queue1, queue2, queue3

shared_events = [
    [("com", queue1), ("arr", queue2)],
    [("arr", queue3), ("com", queue2)]
]
```

Figure 4.4: Example of defining a set of shared events in a three M/M/1 queue tandem network.

simulation cycles, and on which the composition will perform its logic. For semantic convenience, we still use the default keyword for non-synchronised (unique) events.

The `shared_events` property of the `Compose` class is a list of these data structures – see Fig. 4.4 for an example on a tandem network with three M/M/1 queues, each with events `‘arr’` and `‘com’`.

4.4.3 Overloading clock functions

The system makes a tacit assumption that programmer consistently defines the clock functions for shared events. In other words, the `f` and `r` functions always produce the same result (`f` samples the same *distribution*) in all processes which share a synchronised active event.

During simulation loops, rates are computed from the default process with the corresponding state component. Moreover, the `Compose` class does not have a clock mapping, instead it will use the built-in clocks of the existing GSMPs; so, when a value is needed it is obtained directly from the (default) process, conversely event clocks are set and decremented in these processes. Likewise, when a clock distribution must be sampled, the system *tends* to go to default process of new events. The programmer may (incorrectly) assume that given this behaviour, it might be sufficient just to ensure the default process defines these functions correctly, and not to worry if there is some slight inconsistency in the other processes.

New events are shared across a common set of processes with the trigger event, and in the case when a new event is synchronised itself – the default process of the new event may not equal the default process of the trigger event. When this happens, the system performs a search to find a common shared process, from which the distribution function is sampled. This value is then used in setting the new event clock in its default process.

The outlined behaviour is correct with a small set of compositional processes and is kept in for convenience. In general, the compositional GSMP needs to define the clock functions over the shared events – so the class allows *overriding*. When this is done, the system will always go to these global definitions when handling shared events.

Global clock setting functions will have the same argument patterns as those from Sec.3.2.1, however the main difference is that they are called with state vectors, and the events will be *default* and be in tuple form - therefore the programmer must be wary of the definitions they put forth - see Fig.4.5.

4.4.4 Core implementation

On initialisation, the composition class stores a mapping from each process instance to its relative index in the state vector. Thus, any time the system calls functions from each process, it can quickly lookup and extract the relevant state data from the global vector.

Events are wrapped in an internal `Event` class at the base GSMP level; these classes implement compositional equality and have fields to track their parent process, and name. When the `Compose` object begins simulation, it will traverse the shared events given by the programmer, and then replace all the non-default instances with that of the shared event. Furthermore, the shared event is flagged as `shared` and a HashMap is stored within that tracks the event's aliases in its non-default processes. This way the same `Event` object is used by all its sharing processes - therefore we can just use the existing clock from the default process to get the event's lifetime. At this stage, if the model has been supplied with an overridden `f_0`, any initial clocks of shared events are set here.

`Compose` obtains the current state by vectorizing the current state from each component. To probabilistically determine the next state given a trigger event, we only go to the affected pro-

```python
from queues import queue1, queue2, queue3
from numpy.random import exponential


def f(new_state, new_event, old_state, trigger_event):
    if new_event == ("com", queue1):
        return exponential(0.5)
    elif new_event == ("arr", queue3):
        return exponential(0.5)
    else:
        return None     # unreachable


def r(state, event):
    return 1


def f_0(state, event):
    if state == (0, 0, 0) and event == ():
        return 1
    else:
        return 0
```

Figure 4.5: Example of defining the clock functions for a three M/M/1 queue tandem network.

cesses – which are immediately known the corresponding `Event` object – and they choose their next internal state. This means that each process can make use of its existing optimisations (`adjacent_states` and memoization), and the composition never needs to do any more work with these features - for example we never need to take the cartesian product of adjacency lists nor explicitly calculate any probability values for state vectors.

The condition for active and old events in the global scope is that they must be active/old respectively in all their sharing processes - Sec.4.3. By joining all the active/old events from every process, instances of shared events are counted – and when that count differs from the expected value the event is removed from the output. This is performed on every simulation cycle, and allows the model to correctly determine the active, old and new events at all points in time.

Chapter 5

Evaluation

This chapter documents the testing process on the Python library, we first verify correctness by solving the models given in Sec.2.3 and cross referencing our results with the theory. We then test the runtime performance of simulation, both for composition and normal GSMP solving. These are conducted on the 'worst-case' compositional model in the sense that we are testing models with *independent* components. We assess the performance of the compositional and standard GSMP approach for a tandem network by varying the number of queues - thus showing how runtime scales with components. Finally, we evaluate some of the qualitative features of the implementation; we discuss the consequences of building the library in Python, and some of the practical issues that come from our design choices.

5.1 Testing

The results from Sec.2.3 are used to verify accuracy and performance of the software. The queue networks are created as GSMP models, and plugin objects are built to track the performance metrics of simulation.

The models are simulated over a range of queue utilisations by fixing the arrival rate and varying the service rates. We perform a hypothesis test by computing confidence intervals of each test and assessing the accuracy.

5.1.1 M/M/1

We solve the M/M/1 implementation – given in Appendix.A.1 – by simulating over a range of times (differing by orders of magnitude), with the aim of showing that the long run results converge towards the analytic steady state. Mean response and waiting times, and total population and queue sizes are computed from a set a test runs – and the confidence intervals are plotted with the steady state solutions.

The simulation produces a timeline of arrivals and completions, and the response time of the ith job can be calculated as the difference of the ith completion and the ith arrival. If we also track the service times, the waiting time of this job may be taken as the difference between its response and service times. Moreover, all arrivals and completions are independent of each other, therefore all the response and waiting times must be as well. By taking all the response and waiting times, if the total completions in the run exceed 30 then by the Central Limit Theorem the sample means will be normally distributed.

The arrival rate for all the trials is fixed to 1, and nine tests are performed over the utilisations $\rho = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$. The service rates are computed by λ/ρ. Runs are performed with no warmup over the simulation times 1000, 10000, 50000; the blue areas show the 95% confidence intervals, and the analytic result is in red. Fig.5.1-5.2 show the mean response and waiting times.

The utilisation beyond $\rho = 0.7$ is considered heavy load, and the confidence intervals grow larger because it takes longer for the system to reach equilibrium – i.e., the queues tend to grow larger because of the slower service. Conversely for light load the simulation converges extremely quickly to the steady state for all three times – it takes a simulation time two orders of magnitude greater to obtain stability in the heavy load systems.

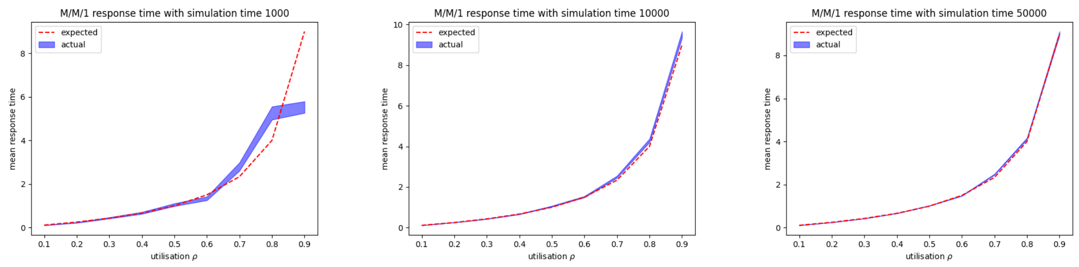

Figure 5.1: M/M/1 mean response times

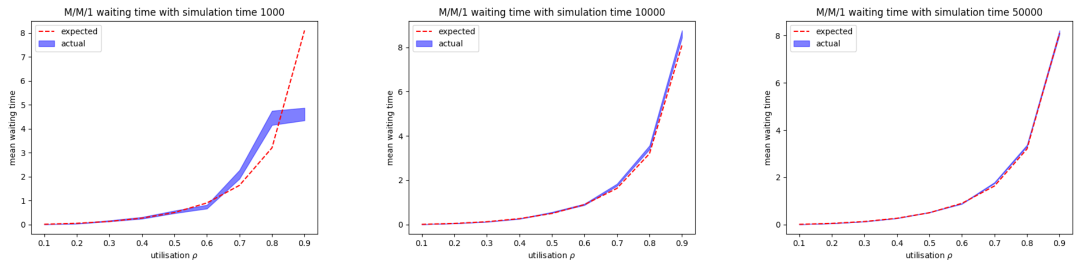

Figure 5.2: M/M/1 mean waiting times

To obtain mean queue and total job sizes, a simulation is run for an extended time, divided into fixed time intervals. At a random point within each interval the state of the queue is sampled. The number of jobs in the system is the exact value, and the queue length can be given by $\max 0, state - 1$ since one customer is always being serviced. Again, the arrival rate is fixed to 1, a single run is performed for 10000-time units with 95% confidence intervals. Changing the size of the sampling intervals will affect the size of the confidence intervals and therefore change the accuracy – see Fig.5.3-5.4.

Due to the lower sample size relative to the response and waiting time results, the confidence intervals are much larger, and the increase in batch count demonstrates that the populations are converging towards the steady state.

The former results demonstrate that M/M/1 queues stabilise in long-run simulations, and the expected steady state values lie within 95% confidence intervals of the sample means.

Finally, we compare the steady state distribution of M/M/1 to M/M/1/PS - see Appendix.A.2 for the Python code, by fixing $\lambda = 1$, and varying utilisations $\rho = 0.2, 0.5, 0.8$ – see Fig.5.5. The results are computed over times 100, 1000, 10000 with no warmup, and show that in both cases extending the simulation period results in better approximations to the true pdf.

For the light load, the systems converge within small simulation time of 100 units; conversely

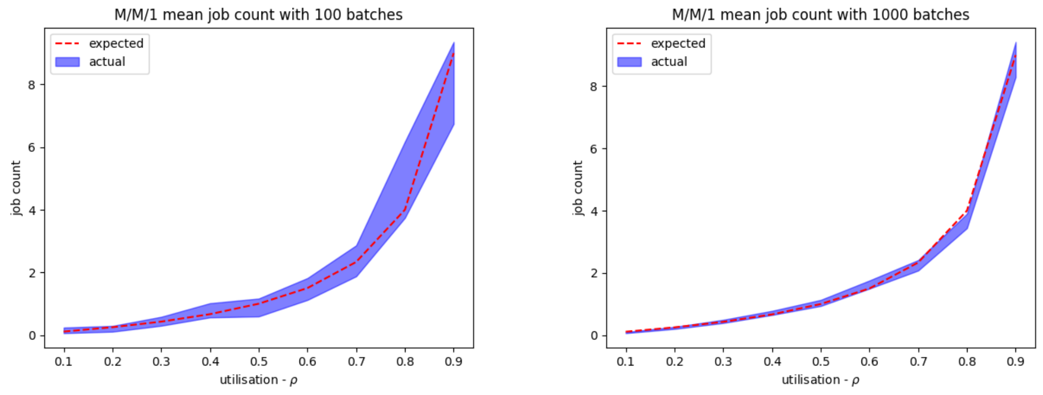

Figure 5.3: M/M/1 mean population size

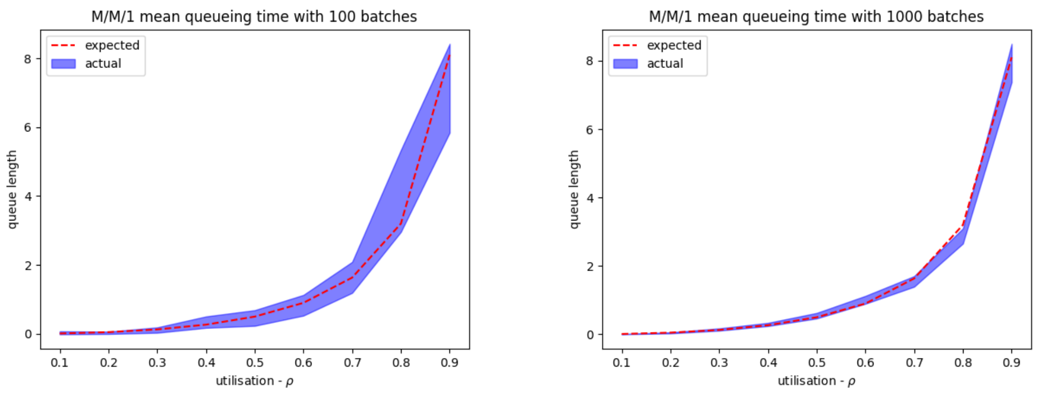

Figure 5.4: M/M/1 mean queue size

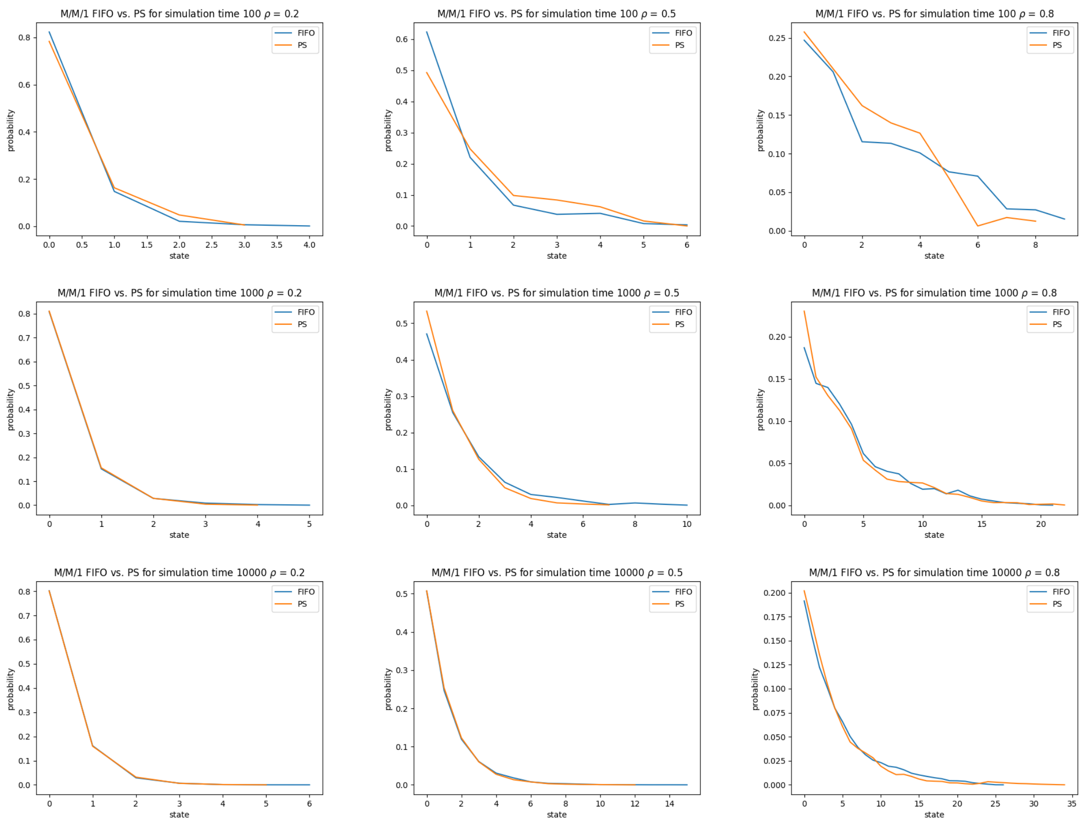

Figure 5.5: Probability density functions of M/M/1 and M/M/1/PS for varying simulation times and utilisations.

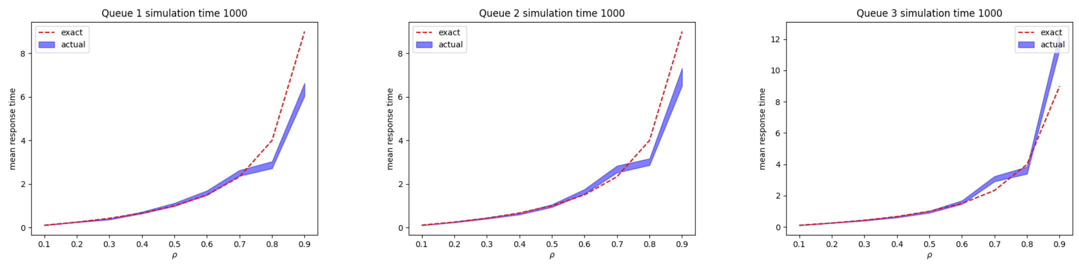

Figure 5.6: Mean response time for three tandem queues with simulation time 1000 units.

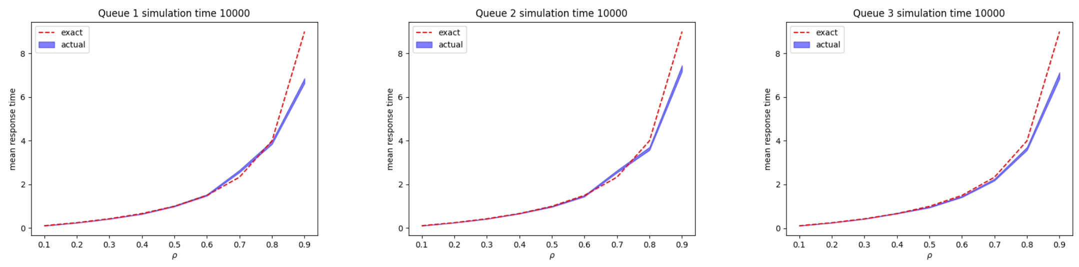

Figure 5.7: Mean response time for three tandem queues with simulation time 10000 units.

the heavy load queues are very unstable. For the 10000-time run, the heavy load pdfs are stabilising and converge towards the same values.

5.1.2 Tandem network

For brevity we solve a tandem system with three M/M/1 queues and demonstrate that the queues are independent by producing sample probability distributions of each queue. These pdfs are compared the steady state solution of a single M/M/1 queue. Moreover, because the theory shows that the queues are independent, it follows that the performance metrics of each queue will be as well – and they should converge towards the results in Sec.2.3.1.

Tests are conducted by fixing the arrival rate to 1 and varying the utilisation of all three queues at the same time; furthermore, each of the tests are simulated over times 1000, 10000, and 100000 units to demonstrate how the solutions accuracy improves.

Mean response data is provided in Fig.5.6-5.8 respectively for all three queues using composition. Increasing the simulation times brings the mean response times closer towards the steady state values (red), and the 95% confidence intervals (blue) are converging. The results from each queue are very similar to those from the previous section.

To compute the pdf of this system, 10 independent sample runs are performed, each generating a stateful pdf for each queue. The values are averaged, and 95% confidence interval are determined; then these intervals are plotted (in blue) with the steady state pdf (for all three queues). This is repeated for utilisation $\rho = 0.2, 0.5, 0.8$, and for simulations times 1000, 10000, and 100000. The results are shown in Fig.5.9-5.11.

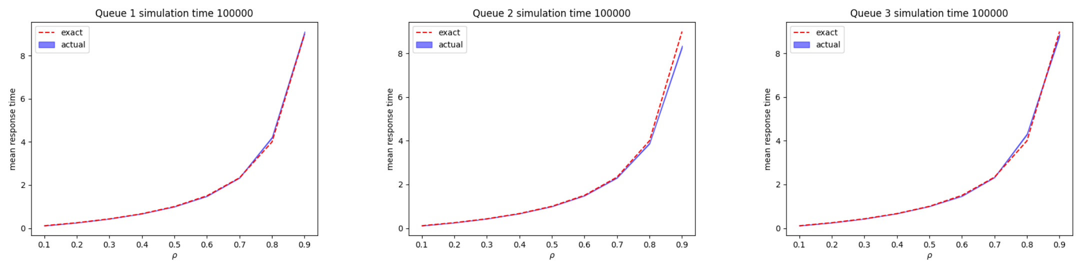

Figure 5.8: Mean response time for three tandem queues with simulation time 100000 units.

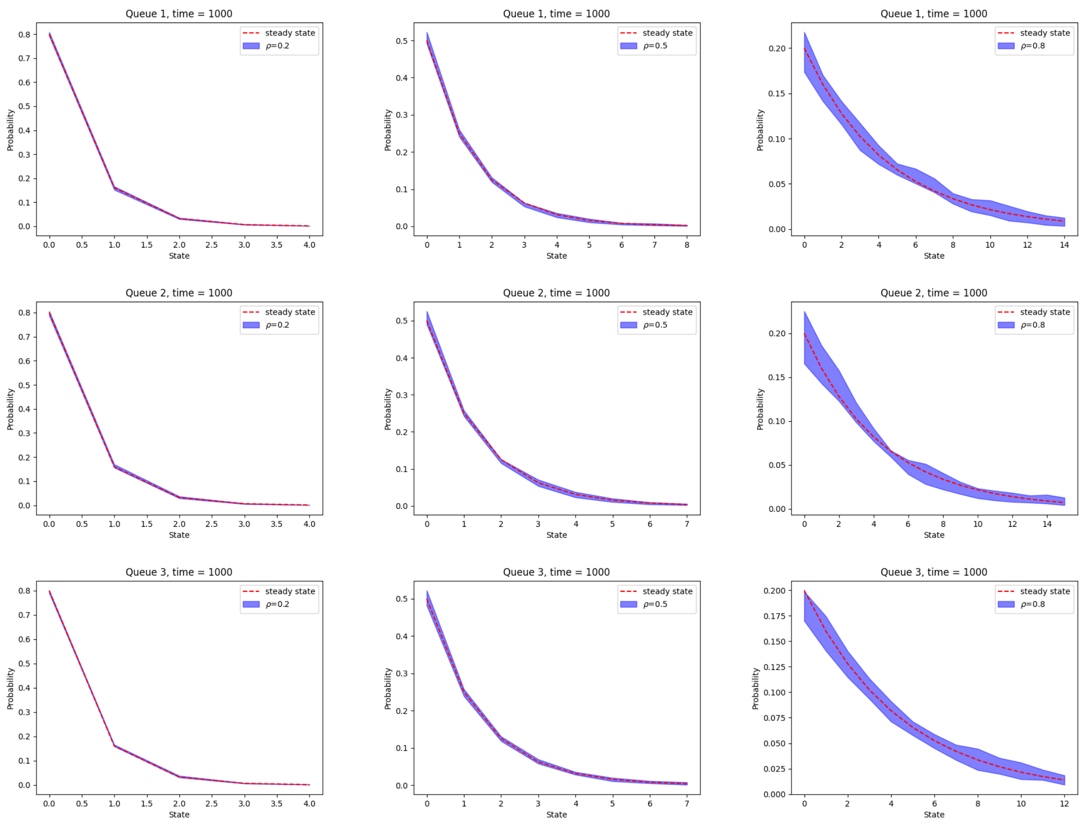

Figure 5.9: Tandem network pdfs for simulation time 1000 units

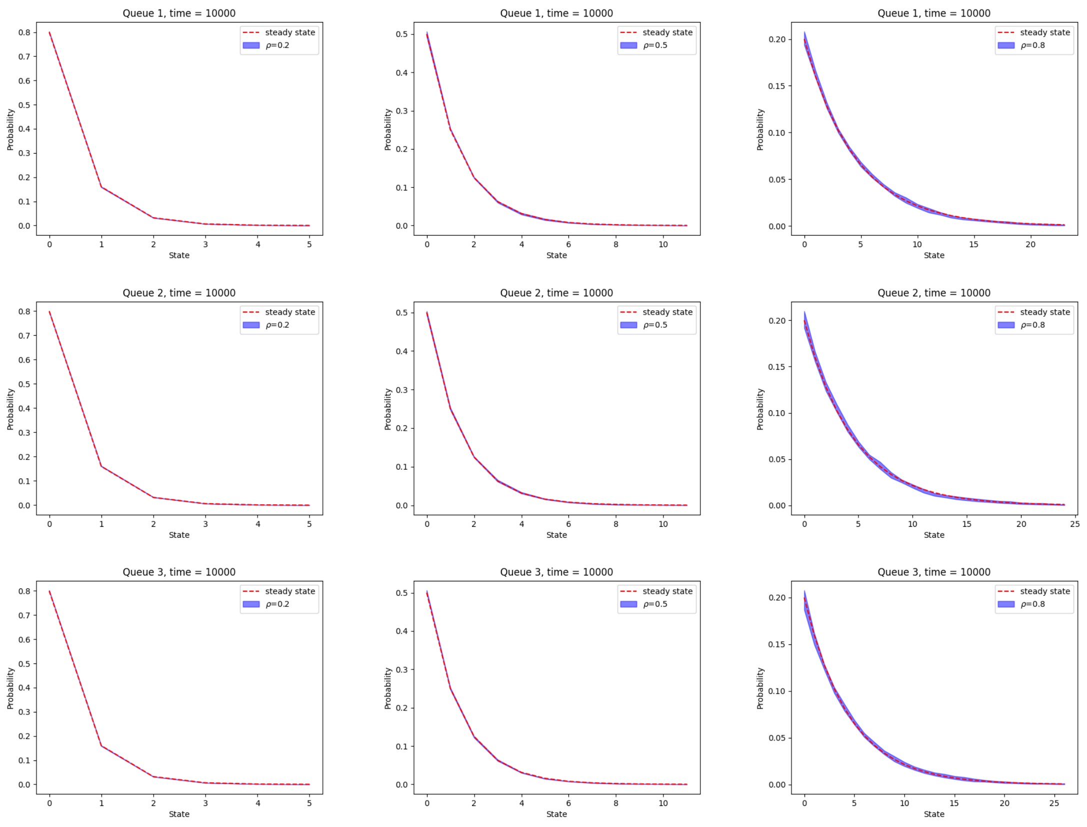

Figure 5.10: Tandem network pdfs for simulation time 10000 units

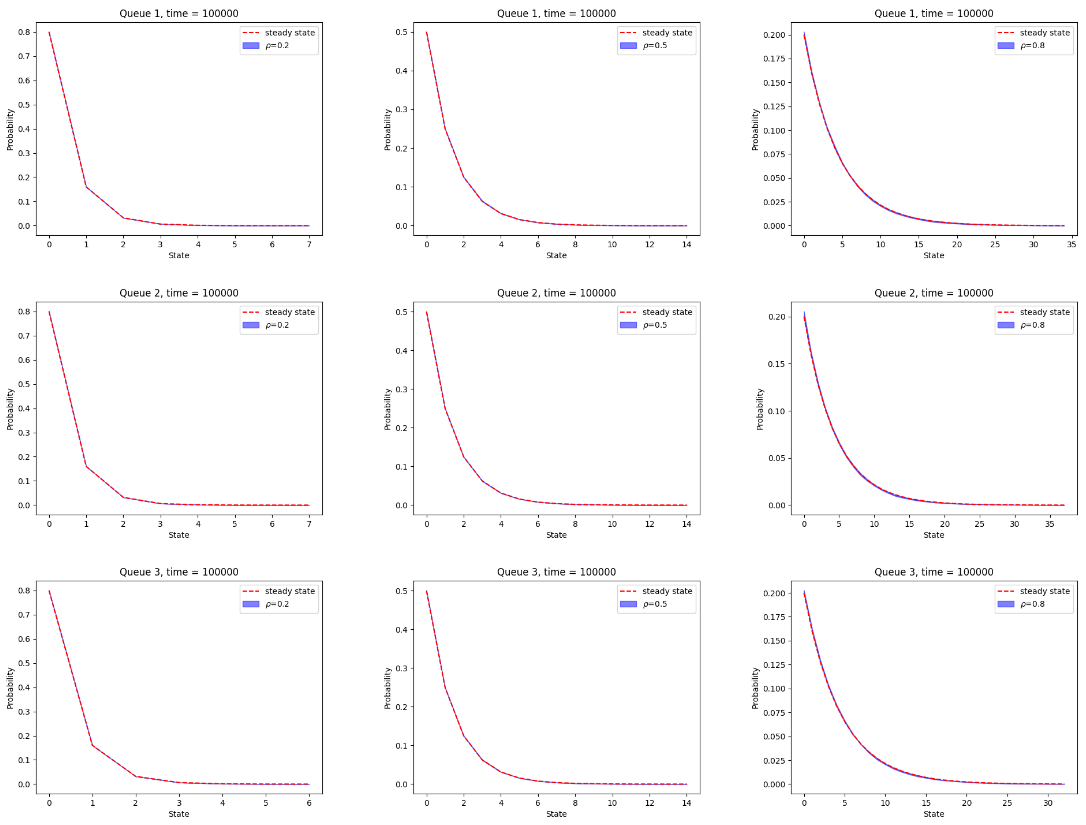

Figure 5.11: Tandem network pdfs for simulation time 100000 units

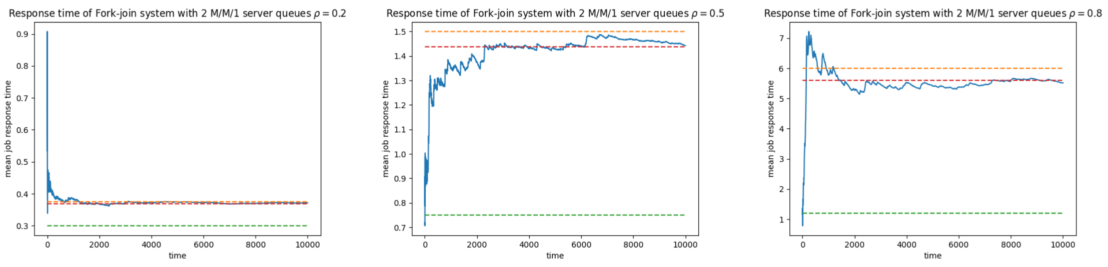

Figure 5.12: Fork join response with two parallel M/M/1 queues.

5.1.3 Fork-join response times

We model a Fork-join system with k parallel M/M/1 servers, each with arrival rate $\lambda = 1$, service rate μ and utilisation $\rho = \frac{\lambda}{\mu}$. The queues are composed and synchronised over their arrival events, results are collected for $k = 2, 4, 8, 16, 32$ and $\rho = 0.2, 0.5, 0.8$. From these utilisations the service rates are determined. All simulations are run for simulation time of 10000 units with no warmup.

A plugin determines the response times at each queue and works out the overall response once all the queues have completed. The mean response time is graphed over the entire time interval, and the results are compared to the upper (orange) and lower (green) bounds, and the approximate value. See Figs.5.12-5.16. The Nelson and Tantawi approximation (Sec.2.3.3)values lie (red) within the 99% confidence interval, which for all test cases fall within 2% of the sample mean.

5.2 Composition time performance

In this section we analyse the runtime performance of composition, namely the effect of increasing the number of components, and comparison to regular GSMP modelling. Simulations are run for time intervals 1000, 2000, . . . , 10000 units, and averages are collected from ten experiments. The

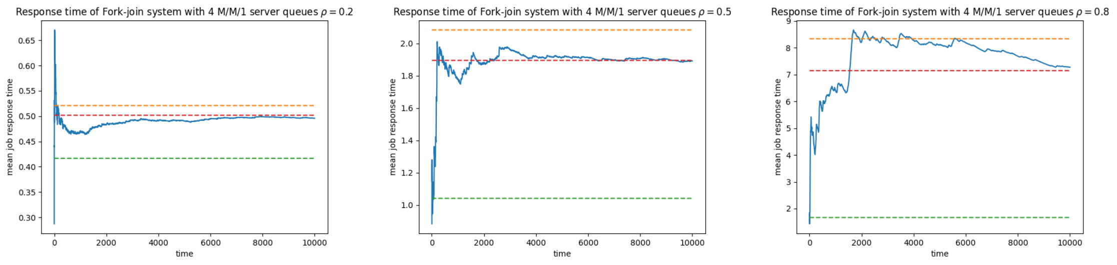

Figure 5.13: Fork join response with four parallel M/M/1 queues.

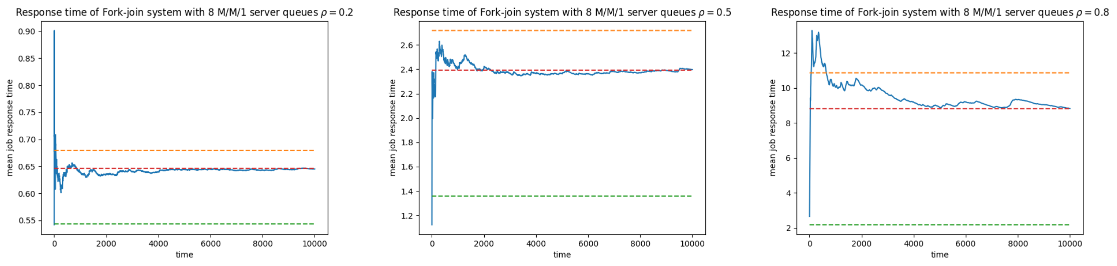

Figure 5.14: Fork join response with eight parallel M/M/1 queues.

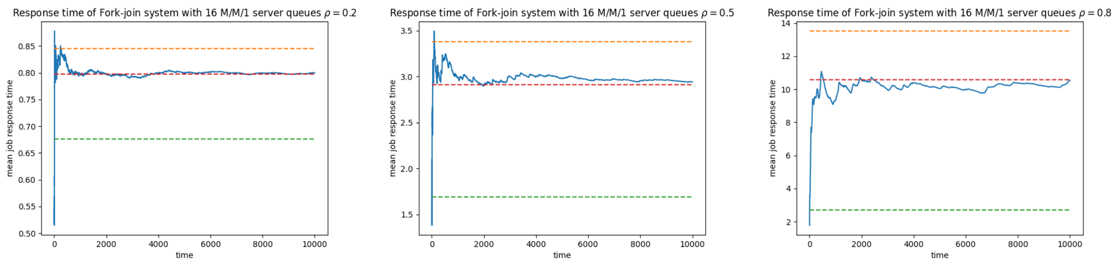

Figure 5.15: Fork join response with sixteen parallel M/M/1 queues.

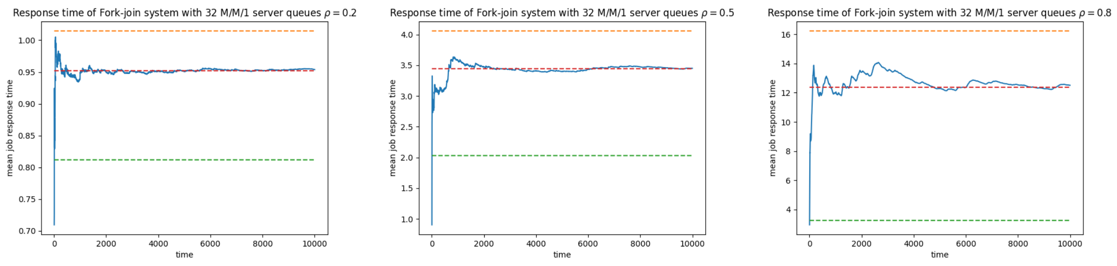

Figure 5.16: Fork join response with thirty-two parallel M/M/1 queues.

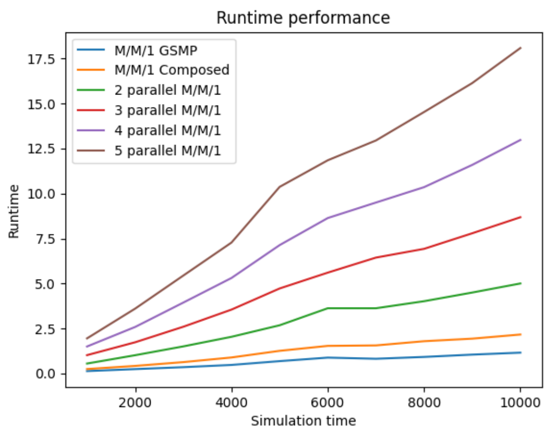

Figure 5.17: The relationship between simulation time and runtime for compositional models.

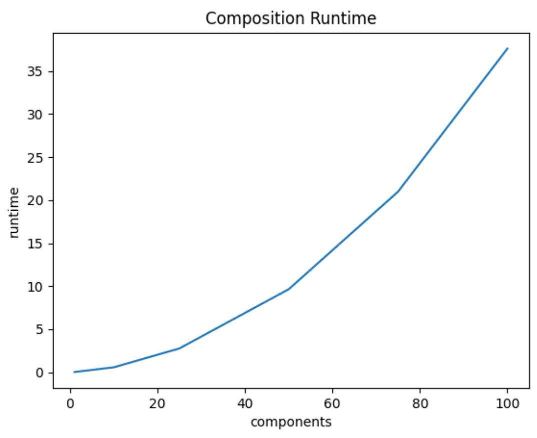

Figure 5.18: n independent parallel M/M/1 queues are simulated in composition for 100 units.

first system will be a sequence of parallel M/M/1 queues, from one to five. For the single queue, results are obtained for a standard GSMP simulation, and another in which the queue is wrapped in the Compose object – Fig.5.17. All of the M/M/1 queues tested are configured $\lambda = 1$, $\mu = 2$.

The runtimes are proportional to the simulation time; however, the slopes increase with the number of parallel components. Nevertheless, the actual *gap* between the composition runtimes seems equal. Consider the case between the modelling of a single M/M/1 queue, and 5 parallel queues. Over the same simulation period, we would expect the latter to have 5 times as many events and state transitions per unit time – we refer to this as **activity** for the rest of this section.

As the runtime scales linearly with simulation time, it stands to reason that for the rest of this section, we may conduct test over a small simulation time, and that the results can be interpolated for longer simulation. Now we plot a similar parallel system – with components $n = 1, 10, 25, 50, 75, 100$ over a fixed simulation of 100 units. 10 sample runs are performed for each, and the means are plotted in Fig.5.18. Contrary to intuition, the runtime performance is actually $O\left(n^2\right)$.

Why does this happen? Python has a built-in profiler – a tool for tracking function calls and runtimes – which can be used to find performance bottlenecks. Here we profile a single run of the system for $n = 50, 100$ (Fig.5.19-5.20 respectively, the columns of interest are ‘`ncalls`’ and ‘`cumtime`’) that reveals that the call the number of calls to all the higher-level path generation functions is proportional to activity (twice as many parallel queues will have double the activity). But the actual runtimes are about four times as large in the latter case, contrary to the activity expectation. For each function call, the system needs to iterate through all components – hence the quadratic runtime.

Runtime is conditioned on activity and components; however, the activity is determined by

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
99933	0.934	0.000	3.496	0.000	core.py:452(get_current_state)
199865	0.592	0.000	45.691	0.000	core.py:472(get_active_events)
99932	2.468	0.000	33.193	0.000	core.py:502(choose_winning_event)
99932	1.405	0.000	10.353	0.000	core.py:489(get_new_state)
99932	8.530	0.000	66.029	0.001	core.py:525(set_old_clocks)
99932	0.524	0.000	54.580	0.001	core.py:536(set_new_clocks)
99932	1.580	0.000	2.524	0.000	core.py:518(set_current_state)

Figure 5.19: Profiler results for parallel run of 50 M/M/1 GSMPs over simulation time 1000 units.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
200109	3.180	0.000	12.912	0.000	core.py:452(get_current_state)
400217	1.349	0.000	171.126	0.000	core.py:472(get_active_events)
200108	9.275	0.000	126.710	0.001	core.py:502(choose_winning_event)
200108	4.928	0.000	27.998	0.000	core.py:489(get_new_state)
200108	32.922	0.000	251.762	0.001	core.py:525(set_old_clocks)
200108	1.354	0.000	202.686	0.001	core.py:536(set_new_clocks)
200108	5.684	0.000	9.097	0.000	core.py:518(set_current_state)

Figure 5.20: Profiler results for parallel run of 100 M/M/1 GSMPs over simulation time 1000 units.

the independence of the components. If they are all independent, as in former example, then the activity of the system is at its *highest* - which means we have the *worst case* scenario; when components share events, the activity falls because the system introduces blocking (of inactive events) which means that, on average, fewer function calls will be made per cycle. Thus, the $O\left(n^2\right)$ overhead is an upper bound on runtime performance. Note that this is actually a pitfall of the design approach - whilst we are modelling a *concurrent* system, the simulation of each queue is performed in a *single* thread.

But how much overhead does the actual composition operation bring compared to a regular GSMP modelling approach. In Fig.5.17 the composed M/M/1 has longer runtimes than the standard – albeit the margins are much finer than the other curves. This occurs because python has relatively high function call overheads, which aggregate over the longer simulation times, however a more sensible comparison would be in the context of a compositional model – specifically the simulation of tandem M/M/1/10 queues. For this test, we perform 10 sample runs for networks of $n = 2, 3, 4, 5$ queues, all with utilisation $\rho = 0.5$. The mean runtimes are plotted in Fig.5.21 for the a composition (with models following directly from Fig.4.3) and an explicit GSMP - see Appendix.A.3 for the implementation of the latter.

The compositional approach scales well with the number of components and falls within the $O(n^2)$ bound established above. Unfortunately the explicity tandem queue GSMPs have runtimes

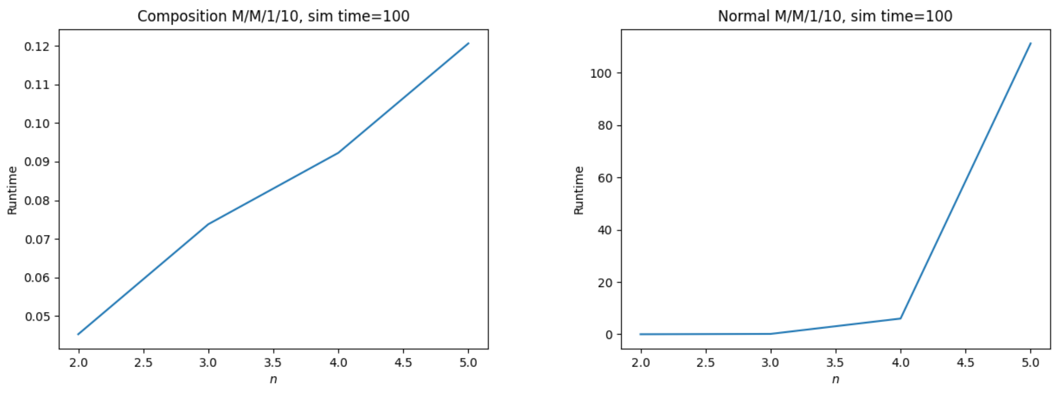

Figure 5.21: Short simulation of $n = 2, 3, 4, 5$ M/M/1/k queue Tandem network with both GSMP approaches.

exponential in n - notice the difference in the scale; the performance for $n = 4, 5$ are orders of magnitude greater than the pure composition.

Note that here there is no `adjacent_states` optimisation for either case (but we still have caching): therefore when the composition finds adjacent states it only needs to look at $10n$ different possibilities, whereas the regular GSMP evaluates 10^n! This example is a bit contrived in the sense that a simple formula can be given for adjacent states in the tandem network, namely

$$\texttt{adjacent\_states}(s = (s_1, s_2, \cdots)) = (s_1 \pm 1, s_2 \pm 1, \cdots)$$

.

(With an edge case to prevent -1 from appearing in the vector, perhaps with some min function?) Using this optimisation would actually produce *better* results for the normal tandem queue GSMP than the composition (because of slightly lower internal overheads); however, deducing a generalised way to build efficient `adjacent_states` functions for a GSMP is *not* a trivial problem. The purpose of this test is to verify that, even for simple systems, our composition is *much* more efficient in the naive context.

5.3 Qualitative evaluation

The GSMP interface forms a simple and intuitive domain specific language with the benefit of Python's readability. The ease-of-use extends to composition, which can be achieved in as little as a single line.

Building the software as a Python library levies the many benefits of the language: it is regularly updated and there are many reputable third-party libraries which can be integrated into user programs. Many complicated tasks are abstracted at a very high level, for instance NumPy is used to efficiently sample exponential distributions in the GSMP models – see appendix. The dynamic typing and syntactic sugaring make it very easy to learn; conversely there is a lot of scope for error. As a high-level language, the default CPython interpreter tends to produce much longer runtimes than popular languages such as Java. It is significantly slower than C++ and other lower-level languages, nevertheless compelling benefits such as automatic referential garbage collection, comprehension and generators, and the broad built-in libraries mediate the drawbacks.

Using plugins to parse transition data real time brings a lot of versatility; since data is sent after every simulation loop there are many options. Some examples include extracting performance metrics; streaming the data over the internet; and getting real-time results on an indefinite simulation (such as graphs). Moreover, the plugin handler can be run in parallel to the simulation, and with a clever approach all kinds of runtime benefits can be attained. Unfortunately, this flexibility comes at the expense of the user's time and programming skill; they need to design and build the algorithms to handle the data which may require using tertiary libraries.

Generally, the core will assume consistency and well-formedness – thus putting a lot of trust in the programmer. Some conditions are placed on typing, however there are minimal checks in the pre-processing stage, so errors are flagged during runtime. This can be extremely costly in long run simulations of complex models, where a slight programming mistake, such as a malformed pdf, may crash the entire run.

Finally, the system inherently calls the GSMP only with the immediate state history, thus it can solve any time-homogeneous model. The simulation of *any* arbitrary DES can be represented by a time inhomogeneous GSMP, which is conditioned on the entire history; with the use of plugins that save the history, the programmer may feed that data back into their model – thus working around the limitations of the system. Thus, the library can model *every* DES.

Chapter 6

Conclusion

This project has successfully developed a Python library for solving GSMPs, capable of modelling any discrete event system. GSMPs are simple to build, and the tool supports finite as well as countably infinite state-spaces and event sets; furthermore native support for third-party Python libraries makes the GSMP library easy to integrate into other systems. The core GSMP is capable of producing any MP or SMP, and in general time homogeneous GSMPs, but, through the use of plugins, may extend to all GSMPs and hence all DES.

We have explored a novel approach to GSMP composition by reasoning upon the process properties in a simulation context. From this a set of rules was developed for compositional GSMPs which aligns with cooperation in PEPA, and the library was subsequently extend. We have demonstrated that the software is reliable by reproducing the analytic M/M/1 and M/M/1/PS solutions, which shows that normal GSMP modelling is accurate; furthermore, we verify the compositional method by solving tandem and fork-join systems with the library, which converge to the theoretical results.

Compositional GSMP runtimes have been shown grow within $O(n^2)$ as the number of component GSMP rises; which significantly outperforms the exponential counterpart in regular GSMPs for unoptimized systems, thus avoiding the combinatorial explosion. Furthermore, given the ease of composition within the library; we have thus presented a solution to the tractability issues.

There are two types of extensions which can be applied, performance improvement and quality of life changes. Compositional overhead may be reduced and introducing some concurrency features for parallelism in simulation could yield major runtime benefits. Furthermore, adding built-in plugins for performance metrics and synchronisation schemes such as approximate rates for systems, like Markovian queues, alongside the core can make the libraries easier to use. Improving debugging features and implementing further model consistency checks will aid in the user experience. Finally, a key interest area beyond this project would involve generalising adjacency lists for GSMP states. Implementing a polyhedral model into the system to replace the naive built-in state selection could remove combinatorial issues altogether.

Appendix A

Sample code

A.1 M/M/1

This file creates the M/M/1 model, but when run will simulate the queue, and graph the observed pdf. The generator in the `states` function creates an infinite state space. In the constructor we also define an `adjacent_states` function which is used to optimise the program. Note that for models with infinite state spaces such as this, this parameter is necessary for termination.

```python
from core import Gsmp
from numpy.random import exponential

arrival = 1
service = 2


class MM1(Gsmp):

    def states(self):
        s = 0
        while True:
            yield s
            s += 1

    def events(self):
        return ["arr", "com"]

    def e(self, state):
        if state == 0:
            return ["arr"]
        return self.events()

    def p(self, next_state, event, current_state):
        if event == "arr":
            return int(next_state == current_state + 1)
        else:
            return int(next_state == current_state - 1)

    def f(self, next_state, new_event, current_state, winning_event):
        if new_event == "arr":
            return exponential(1 / self._arrival)
        else:
            return exponential(1 / self._service)

    def r(self, state, event):
        return 1

    def s_0(self, state):
        return int(state == 0)

    def f_0(self, state, event):
        return exponential(1 / self._arrival)

    def __init__(self,
                 adjacent_states=None,
                 arrival_rate=arrival,
                 service_rate=service,
                 name=None):
        self._arrival = arrival_rate
        self._service = service_rate
        self._name = name
        if adjacent_states is None:
            def adjacent_states(s):
                return [1] if s == 0 else [s - 1, s + 1]
        super().__init__(adjacent_states=adjacent_states)

    def __repr__(self):
        if self._name is None:
            return hex(id(self))
        return self._name


rho = arrival / service

queue = MM1()

if __name__ == "__main__":
    from core import Simulator
    epochs = 1000
    # generate results
    data = Simulator(queue).run(epochs=epochs, estimate_probability=True)

    # unpack data
    states, observed_probabilities = zip(*data)

    # Expected M/M/1 probabilities
    expected_probabilities = list(map(
        lambda n: (1 - rho) * rho ** n,
        states
    ))

    # Plot pdf
    from utility import print_results
    print_results(
        p=expected_probabilities,
        ys=[(observed_probabilities, "M/M/1")]
    )
```

A.2 M/M/1/PS

The M/M/1/PS queue is simultaneously serving all customers at once, therefore the GSMP has an infinite set of events. Note that even though we have an `adjacent_states` function, we do not need a events equivalent because the GSMP explicitly defines it (in `E(s)`).

```python
from numpy.random import exponential
from core import Gsmp

arrival, service = 1, 2


class MM1PS(Gsmp):

    def states(self):
        s = 0
        while True:
            yield s
            s += 1

    def events(self):
        yield "arr"
        s = 1
        while True:
            yield "com {}".format(s)
            s += 1

    def e(self, s):
        if s == 0:
            return ["arr"]
        else:
            return ["arr"] + ["com {}".format(i + 1) for i in range(s)]

    def p(self, _s, e, s):
        if e == "arr":
            return int(_s == s + 1)
        else:
            return int(_s == s - 1)

    def f(self, _s, _e, s, e):
        if _e == "arr":
            return exponential(1 / self._arrival)
        else:
            return exponential(1 / self._service)

    def r(self, s, e):
        if e == "arr":
            return 1
        else:
            return 1 / s

    def s_0(self, s):
        return int(s == 0)

    def f_0(self, s, e):
        return exponential(1 / self._arrival)

    def __init__(self,
                 service_rate=service,
                 arrival_rate=arrival,
                 adjacent_states=None):
        if adjacent_states is None:
            def adjacent_states(n):
                return [1] if n == 0 else [n - 1, n + 1]
            self._service = service_rate
            self._arrival = arrival_rate
            super().__init__(adjacent_states=adjacent_states)
```

A.3 Tandem Queue GSMP

This file defines tandem networks of $n = 2, 3, 4, 5$ M/M/1 queues, all with fixed utilisations $\rho = 0.5$. Here the core feature set from each queue has been extracted into superclass `Tandem_queue`, and that other classes define the specific parameters for their respective queues. While correct, the tractability issues are highlighted by the increasing complexity of each class of network.

```
from core import Gsmp
```

```python
from numpy.random import exponential
from itertools import product

k = 20
arrival_rate = 1
service_rate = 2
avg_arrival_time = 1 / arrival_rate
avg_service_time = 1 / service_rate


class Tandem_queue(Gsmp):

    def f(self, _s, _e, s, e):
        if _e == "arr":
            return exponential(avg_arrival_time)
        else:
            return exponential(avg_service_time)

    def r(self, s, e):
        return 1

    def f_0(self, s, e):
        return exponential(avg_arrival_time)

    def __init__(self, k=k):
        self._k = k
        super().__init__()


class Tandem_queue2(Tandem_queue):
    def states(self):
        # cartesian product
        return product(range(self._k + 1), repeat=2)

    def events(self):
        return ["arr", "com1", "com2"]

    def e(self, s):
        (x, y) = s
        es = []
        if x < self._k:
            es.append("arr")
        if x > 0 and y < self._k:
            es.append("com1")
        if y > 0:
            es.append("com2")
        return es

    def p(self, _s, e, s):
        x1, y1 = s
        x2, y2 = _s
        if e == "arr":
            return bool(x1 + 1 == x2 and y1 == y2)
        elif e == "com1":
            return bool(x1 - 1 == x2 and y1 + 1 == y2)
        else:
            return bool(x1 == x2 and y1 - 1 == y2)

    def s_0(self, s):
        return int(s == (0, 0))


class Tandem_queue3(Tandem_queue):
    def states(self):
        # cartesian product
        return product(range(self._k + 1), repeat=3)

    def events(self):
        return ["arr", "com1", "com2", "com3"]

    def e(self, s):
        (x, y, z) = s
        es = []
        if x < self._k:
            es.append("arr")
        if x > 0 and y < self._k:
            es.append("com1")
        if y > 0 and z < self._k:
            es.append("com2")
        if z > 0:
            es.append("com3")
        return es

    def p(self, _s, e, s):
        x1, y1, z1 = s
        x2, y2, z2 = _s
        if e == "arr":
            return bool(x1 + 1 == x2 and y1 == y2 and z1 == z2)
        elif e == "com1":
            return bool(x1 - 1 == x2 and y1 + 1 == y2 and z1 == z2)
        elif e == "com2":
            return bool(x1 == x2 and y1 - 1 == y2 and z1 + 1 == z2)
        else:
            return bool(x1 == x2 and y1 == y2 and z1 - 1 == z2)

    def s_0(self, s):
        return int(s == (0, 0, 0))


class Tandem_queue4(Tandem_queue):
    def states(self):
        # cartesian product
        return product(range(self._k + 1), repeat=4)

    def events(self):
        return ["arr", "com1", "com2", "com3", "com4"]

    def e(self, s):
        (x, y, z, a) = s
        es = []
        if x < self._k:
            es.append("arr")
        if x > 0 and y < self._k:
            es.append("com1")
        if y > 0 and z < self._k:
            es.append("com2")
        if z > 0 and a < self._k:
            es.append("com3")
        if a > 0:
            es.append("com4")
        return es

    def p(self, _s, e, s):
        x1, y1, z1, a1 = s
        x2, y2, z2, a2 = _s
        if e == "arr":
            return bool(x1 + 1 == x2 and
                        y1 == y2 and
                        z1 == z2 and
                        a1 == a2)
        elif e == "com1":
            return bool(x1 - 1 == x2 and
                        y1 + 1 == y2 and
                        z1 == z2 and
                        a1 == a2)
        elif e == "com2":
            return bool(x1 == x2 and
                        y1 - 1 == y2 and
                        z1 + 1 == z2 and
                        a1 == a2)
        elif e == "com3":
            return bool(x1 == x2 and
                        y1 == y2 and
                        z1 - 1 == z2 and
                        a1 + 1 == a2)
        else:
            return bool(x1 == x2 and
                        y1 == y2 and
                        z1 == z2 and
                        a1 - 1 == a2)

    def s_0(self, s):
        return int(s == (0, 0, 0, 0))


class Tandem_queue5(Tandem_queue):
    def states(self):
        # cartesian product
        return product(range(self._k + 1), repeat=5)

    def events(self):
        return ["arr", "com1", "com2", "com3", "com4", "com5"]

    def e(self, s):
        (x, y, z, a, b) = s
        es = []
        if x < self._k:
            es.append("arr")
        if x > 0 and y < self._k:
            es.append("com1")
        if y > 0 and z < self._k:
            es.append("com2")
        if z > 0 and a < self._k:
            es.append("com3")
        if a > 0 and b < self._k:
            es.append("com4")
        if b > 0:
            es.append("com5")
        return es

    def p(self, _s, e, s):
        x1, y1, z1, a1, b1 = s
        x2, y2, z2, a2, b2 = _s
        if e == "arr":
            return bool(x1 + 1 == x2 and
                        y1 == y2 and
                        z1 == z2 and
                        a1 == a2 and
                        b1 == b2)
        elif e == "com1":
            return bool(x1 - 1 == x2 and
                        y1 + 1 == y2 and
                        z1 == z2 and
                        a1 == a2 and
                        b1 == b2)
        elif e == "com2":
            return bool(x1 == x2 and
                        y1 - 1 == y2 and
                        z1 + 1 == z2 and
                        a1 == a2 and
                        b1 == b2)
        elif e == "com3":
            return bool(x1 == x2 and
                        y1 == y2 and
                        z1 - 1 == z2 and
                        a1 + 1 == a2 and
                        b1 == b2)
        elif e == "com4":
            return bool(x1 == x2 and
                        y1 == y2 and
                        z1 == z2 and
                        a1 - 1 == a2 and
                        b1 + 1 == b2)
        else:
            return bool(x1 == x2 and
                        y1 == y2 and
                        z1 == z2 and
                        a1 == a2 and
                        b1 - 1 == b2)

    def s_0(self, s):
        return int(s == (0, 0, 0, 0, 0))
```

Bibliography

[1] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* USA: Cambridge University Press, 1st ed., 2013.

[2] J. R. Jackson, "Jobshop-like queueing systems," *Management Science*, vol. 50, no. 12, pp. 1796–1802, 2004.

[3] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *J. ACM*, vol. 22, pp. 248–260, Apr. 1975.

[4] L. Flatto and S. Hahn, "Two parallel queues created by arrivals with two demands i," *SIAM Journal on Applied Mathematics*, vol. 44, no. 5, pp. 1041–1053, 1984.

[5] L. Flatto, "Two parallel queues created by arrivals with two demands ii," *SIAM Journal on Applied Mathematics*, vol. 45, no. 5, pp. 861–878, 1985.

[6] R. Nelson and A. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 739–743, 1988.

[7] J. Hillston, *A Compositional Approach to Performance Modelling.* Distinguished Dissertations in Computer Science, Cambridge University Press, 1996.

[8] J. Bradley, S. Gilmore, and N. Thomas, "How synchronisation strategy approximation in pepa implementations affects passage time performance results," in *Applying Formal Methods: Testing, Performance, and M/E-Commerce* (M. Núñez, Z. Maamar, F. Pelayo, K. Pousttchi, and F. Rubio, eds.), vol. II of *Lecture Notes in Computer Science*, pp. 128–142, Springer-Verlag GmbH, 2004.

[9] P. Glynn, "A gsmp formalism for discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 14–23, 1989.

[10] F. Nilsen, "Gmsim: a tool for compositional gsmp modeling," in *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, vol. 1, pp. 555–562 vol.1, 1998.