Quantitative Analysis of Web Services Using SRMC

Allan Clark, Stephen Gilmore, and Mirco Tribastone

Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland

Abstract. In this tutorial paper we present quantitative methods for analysing Web Services with the goal of understanding how they will perform under increased demand, or when asked to serve a larger pool of service subscribers. We use a process calculus called SRMC to model the service. We apply efficient analysis techniques to numerically evaluate our model. The process calculus and the numerical analysis are supported by a set of software tools which relieve the modeller of the burden of generating and evaluating a large family of related models. The methods are illustrated on a classical example of Web Service usage in a business-to-business scenario.

1 Introduction

Web Services are a popular and effective method of component-based development of distributed systems. Using widely-agreed standards service providers are able to quickly develop flexible assemblies of components to respond to new business demands. Legacy systems can be incorporated using application servers as intermediates which expose the functionality of the legacy system on the network, allowing it to be invoked by a remote service. This might itself have been invoked by another service, allowing these components to be built into complex workflows and managed as either an orchestration or a choreography.

Service providers publish their services in a public registry. Service consumers discover services at run-time and bind to them dynamically, choosing from the available service instances according to the criteria which are of most importance to them. This architecture provides robust service in difficult operational conditions. If one instance of a service is temporarily unavailable then another one is there to take its place.

It is likely though that this replacement is not fully functionally identical. It might have some missing functionality, or it might even offer additional functionality not found in the temporarily unavailable service instance. One reason why differences such as this arise is that new versions of services are released in order to correct errors or add new features. These updates are applied at different times at different sites and therefore it is quite common for different hosts to be running different versions of the software services. Some will be running an older version, others the latest. Even if they are hosting the same version

M. Bernardo, L. Padovani, and G. Zavattaro (Eds.): SFM 2009, LNCS 5569, pp. 296–339, 2009.

of the software then because of different security policies at different sites some hosts will have disabled certain features, whereas others will not have done this because their security policy is more permissive.

Even in the rare case of finding a functionally-identical replacement matters are still not straightforward when non-functional criteria such as availability and performance are brought into the picture. It is very unusual indeed for all of the hosts which offer instances of a service to have identical performance profiles. In contrast, the best practice in virtualisation argues that the hosts should intentionally be heterogeneous (using different processors, memory, caches or disks) in order that not all of them can be affected by a single flaw in a hardware component. Seemingly small modifications such as this can have a vast impact on performance which affects essentially all of the performance measures which one would think to evaluate over the system configuration.

In practice it is very frequently the case that the functionally-equivalent replacement for the temporarily unavailable service will exhibit different performance characteristics. Ultimately this is because it hosts a copy of the service on another hardware platform which has either been intentionally made different for reasons such as virtualisation practice, or unintentionally because it has been commissioned at a different time when other hardware components were the most cost-effective purchase.

Analytical or numerical performance evaluation provides valuable insights into the timed behaviour of systems over the short or long run. Important methods used in the field include the numerical evaluation of continuous-time Markov chains (CTMCs) (see, for example, [1]) and the use of fluid-flow approximation using systems of ordinary differential equations (ODEs) (see, for example, [2]). In the present paper we work with a timed process calculus, the Sensoria Reference Markovian Calculus (SRMC) [3,4] which builds on Performance Evaluation Process Algebra (PEPA) [1]. PEPA has both a discrete-state Markovian semantics and a continuous-state differential equation semantics. We make use of both kinds of analysis here.

Mathematical modelling formalisms such as CTMCs and ODEs are often applied to study fixed, static system configurations with known subcomponents with known rate parameters. This is far from the operating conditions of service-oriented computing where for critical service components a set of replacements with perhaps vastly different performance qualities stand ready to substitute for components which are either unavailable, or the consumer just simply chooses not to bind to them.

We seek to address this issue with SRMC by building into the calculus a mechanism for the formal expression of uncertainty about binding and parameters (in addition to the other dimension of uncertainty about durations modelled in the Markovian setting through the use of exponentially-distributed random variables). We put forward a method of numerical evaluation for this calculus which scales well with increasing problem size to allow precise comparisons to be made across all of the possible service bindings and levels of availability considered.

Numerical evaluation is supported inside a modelling environment for the calculus. In addition to comparing the results of particular service configurations we can combine the results to provide overall performance characteristics such as are required for service level agreements.

Structure of this paper. SRMC allows three levels of uncertainty; uncertainty as to the configuration of the system, uncertainty as to the rate parameters of some system components and finally uncertainty as to the duration of events. After an introduction to the calculus in Section 2 we build up to the full SRMC language in reverse order of these levels of uncertainty. In Section 2.2 we review the PEPA process algebra, a stochastic process algebra with support for compositional construction of an underlying Markov chain. Thus we can reason about the performance of a known system with unknown duration of events. We continue in this section to show how we can augment this process algebra with the ability to specify a range of rate parameters such that not only is the duration of a particular event unknown but its average duration is specified as a set of possible values. Because of this a single model in the SRMC calculus gives rise to a related family of models in the PEPA stochastic process algebra. In Section 4 we explain how this family of models is derived. Our intention is to perform analysis on these models. In Section 5 we present a high-level query language for models, eXtended Stochastic Probes (XSP). We show how this language is used to query models to determine whether or not they satisfy precise service-level agreements on their quality of service. In Section 6 we apply Markovian analysis techniques to all of the models in this related family. In Section 7 we address the challenge of large-scale modelling and recast the modelling problem in the continuous world where we can apply Hillston's fluid-flow approximation method [2] to obtain a system of ordinary differential equations which allow us to efficiently analyse large-scale versions of our models. In Section 8 we consider the suite of software tools which are available to support the SRMC and PEPA process calculi. Section 9 surveys related work and we present our conclusions after this.

2 Background

In order to introduce the concepts of SRMC we build up a generic example of a Web Service. We will provide a specific example later.

2.1 SRMC

In this example we have a service which remains idle until it receives a request from a client. The service does not specify the rate at which requests arrive, this is specified elsewhere (in the definition of the client). Once a request comes in the service computes (at rate `r_c`) and then returns the response (at rate `r_r`) before becoming idle again.

Listing 1.1. SRMC model of a Web Service

```
WS_A::{
  r_c = 10.0;  r_r = 1.0;
  Idle       = (request, _).Computing;
  Computing  = (compute, r_c).Responding;
  Responding = (response, r_r).Idle;
};
```

This high-level model of the service describes only three states, `Idle`, `Computing` and `Responding`, abstracting from many details of the service. These three related definitions are collected into the namespace for the component `WS_A` together with the values of the rates for the activities `compute` and `response`.

This definition gives rise to a small transition system with only three states and three transitions. The transition system corresponding to the component `WS_A` is shown in Figure 1. Note that component names and rate names have been replaced by their fully qualified versions. Activity type names (such as `request`, `compute` and `response`) are not subject to this expansion because these names are used to define synchronisation points with other components (and therefore cannot be renamed).

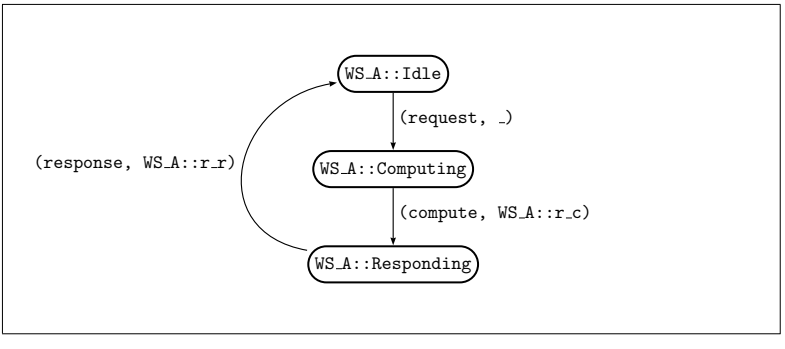


Fig. 1. Underlying transition system for the component `WS_A`

We now consider an optimised version of this service where some computation is avoided because the service can retrieve a previously computed result. Looking up a result is ten times faster than re-calculating it. Only 30% of incoming requests can be answered in this way, the remaining 70% of requests lead to the result being computed as before.

Model components with similar names can be distinguished because they are collected under a different namespace `WS_B`. Thus here we have the definition of a process term whose fully qualified name is `WS_B::Idle` whereas the fully qualified name of the previous process term is `WS_A::Idle`.

Listing 1.2. SRMC model of an optimised Web Service

```
WS_B::{
  r_l = 100.0;  p = 0.7;  r_c = 10.0;
  r_r = { 1.0, 0.6, 0.2 };
  Idle       = (request, p * _).Computing;
             + (request, (1 - p) * _).Retrieving;
  Computing  = (compute, r_c).Responding;
  Retrieving = (lookup, r_l).Responding;
  Responding = (response, r_r).Idle;
};
```

The advantage of using this optimised version of the service is reduced slightly because connectivity to the service is very variable and responses coming back from the service may be delayed (even though the service generated them quickly by looking up a previously-calculated result). The transition system corresponding to the component `WS_B` is shown in Figure 2.

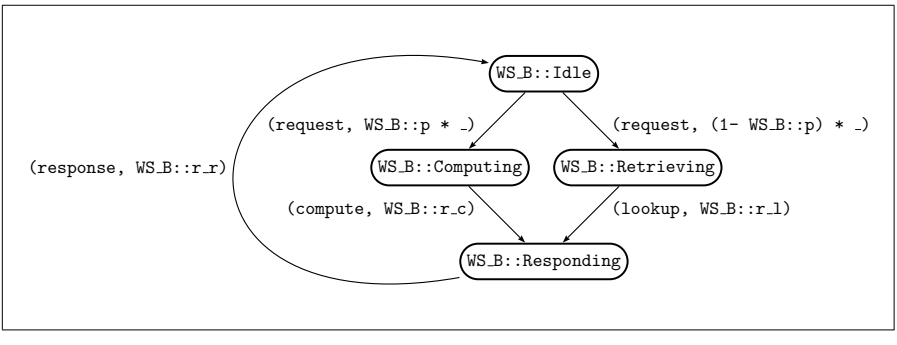


Fig. 2. Underlying transition system for the component `WS_B`

In SRMC we can characterise this kind of variability by recording different possible parameter values for the `response` activity. We denote these by listing a set of possible values for the rate parameter ($\{ 1.0, 0.6, 0.2 \}$ above). Uncertainty about a rate parameter is represented in SRMC in this way (by listing a set of possibilities) and uncertainty about a service binding is represented in a very similar way.

Listing 1.3. Specifying binding uncertainty in SRMC

```
WS ::= { WS_A, WS_B };
Service = WS::Idle;
```

These definitions record that the Web Service which we use will either be A (the unoptimised version) or B (the optimised version) and that the service is initially in its idle state.

Now we are able to complete our model by providing the definition of a client who thinks for some time before requesting the service and waiting for the response.

Listing 1.4. SRMC model of a Client

```
Client::{
  r_t = 0.002;  r_r = 0.5;
  Idle       = (think, r_t).Requesting;
  Requesting = (request, r_r).Waiting;
  Waiting    = (response, _).Idle;
};
```

The namespace mechanism is helpful here also because there is no clash between the name of the rate identifier used here for requests (whose fully qualified name is `Client::r_r`) and rate identifiers used earlier for responses (whose fully qualified names are respectively `WS_A::r_r` and `WS_B::r_r`). The transition system corresponding to the client is shown in Figure 3.

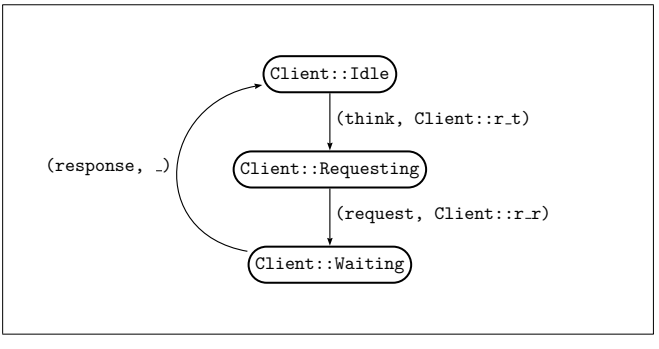


Fig. 3. Underlying transition system for the client

Finally, we complete the model by requiring the Client and the Web Service (whichever one it is) to cooperate on the `request` and `response` activities. All other activities are performed by a single component independently from the others.

Listing 1.5. SRMC model composition

```
Client::Idle <request, response> Service
```

In the case where the binding is resolved in favour of WS_A then the overall model has the transition system corresponding to the client paired with WS_A as shown in Figure 4. All of the definitions which relate to the WS_B namespace are removed from this model and have no impact on the underlying transition system.

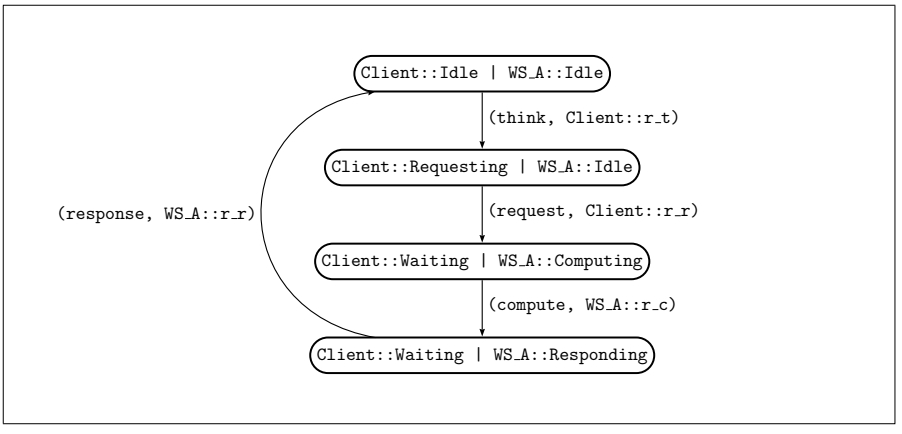


Fig. 4. Underlying transition system for the client paired with WS_A

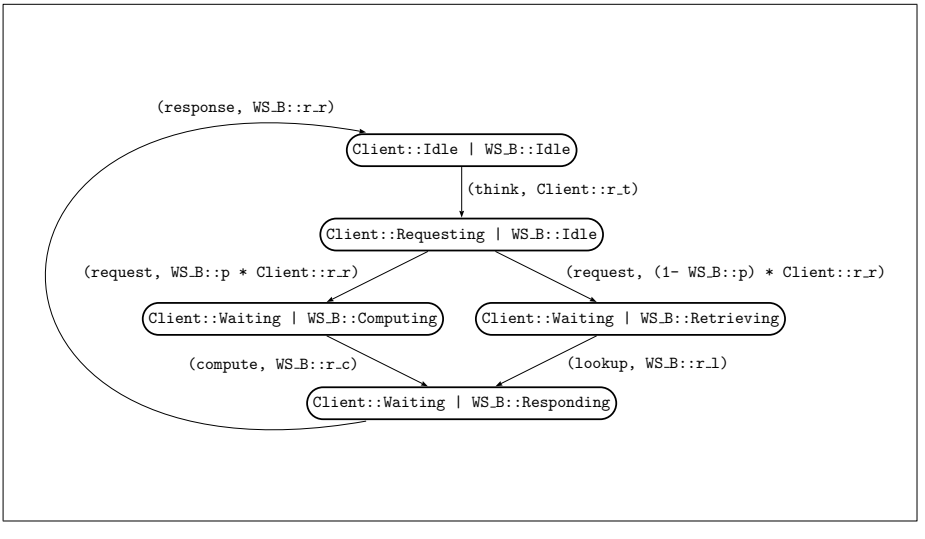


Fig. 5. Underlying transition system for the client paired with WS_B

If, on the other hand, the binding is resolved in favour of WS_B then the overall model has the transition system corresponding to the client paired with WS_B as shown in Figure 5. All of the definitions which relate to the WS_A namespace are removed from this model.

2.2 PEPA

The starting point for the new calculus was the stochastic process algebra PEPA [1]. The PEPA language has the following combinators:

$$P ::= (a, \lambda).P \mid P + P \mid P \underset{L}{\bowtie} P \mid P/L$$

A $(a, \lambda).P$ describes a process which may perform the action a at rate λ to become the process P. The rate may be a numerical rate or the special rate $\top$ (written as _ in SRMC) which means the operation is performed passively by this process which must be subsequently synchronised with over this activity. The other process involved in the synchronisation determines the rate of the activity. The process $P_1 + P_2$ depicts competitive choice between the processes P_1 and P_2 and therefore may perform any of activity which P_1 may perform or any which P_2 may perform. The operator $\underset{L}{\bowtie}$ is cooperation/synchronisation between two components over the given set of actions L. The process P/L behaves exactly as P except that the activities in the set L are no longer observable and hence it is not possible for another process to cooperate on these activities. This is referred to as *hiding*, and we will not make use of hiding in this tutorial.

A model is represented by a series of definitions which describe the sequential behaviour of named components. These named components are then combined together in a main system equation which represents the interaction between the various components in a model. This description of a model has an underlying Markov chain representation though the user is hidden from the details of the underlying states. Each defined sequential component is a description of a small stateful process and each such is combined using the cooperation combinator with the restriction that the two must synchronise over the specified action labels. This may mean that some states are unreachable so composition does not always increase the state space but in general the state-space size does increase rapidly. Full details of the PEPA stochastic process algebra can be found in [1].

3 Case Study

In this tutorial the SRMC language is illustrated by means of a case study of a Web-service orchestration. Our case study is adapted from the example proposed in the specification of WS-BPEL 2.0, the OASIS standard for the description of business process behaviour of Web services [5]. Figure 6 depicts an informal outline of the business process of a sample order management system. Boxes with straight corners represent the invocation of Web services. Dotted lines impose sequentiality between invocations, and solid lines indicate data dependency. For example, the invocation of *Complete Price Calculation* does not start until *Decide On Shipper* returns. Similarly, *Complete Production Scheduling* needs the output of *Arrange Logistics* before being called. The box with rounded corners indicates the execution of parallel flows. After *Receive Purchase Order* is executed, the executions of *Initiate Price Calculation*, *Decide On Shipper*, and *Initiate Production Scheduling* may start in parallel. Finally, after all these activities terminate *Invoice Processing* may be invoked.

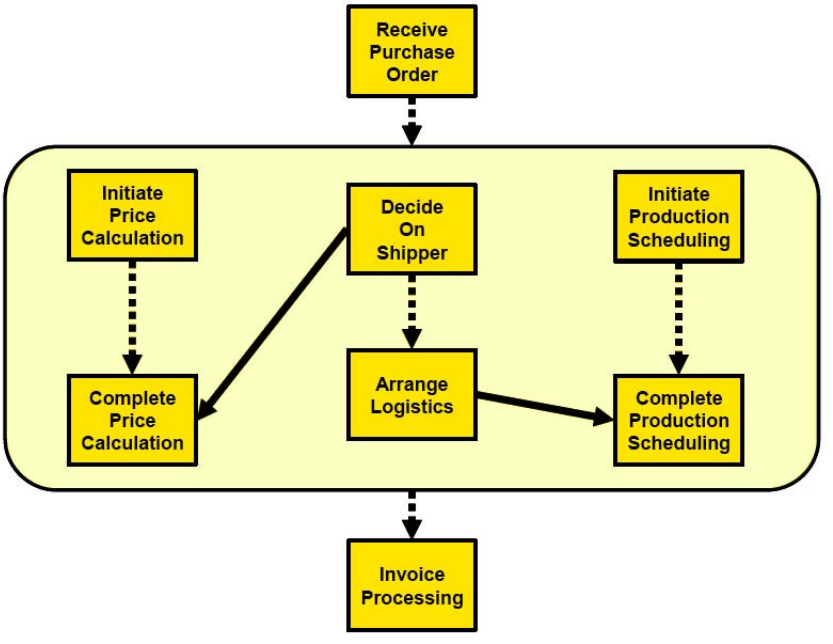


Fig. 6. Sketch of the order management system case study used in this tutorial

The aim of this tutorial is not to provide an algorithmic procedure to automatically translate BPEL processes into SRMC models, rather we show how the features of the language may be exploited to model web service orchestration. Nevertheless, the tutorial will also give directions on how to capture other, more fine-grained behaviours which are not strictly in the domain of web service description languages albeit they may affect the system's performance. The construction of the performance model of the case study will be carried out incrementally — from the components which exhibit sequential behaviour to their arrangement through the cooperation operator to impose synchronisation and ordering. The initial SRMC model will be kept intentionally simple — it will not capture dynamic binding or rate uncertainty, in effect making it fall within the realm of PEPA. This initial model will primarily serve the purpose of guiding the reader through the most basic constituents of the language. Nevertheless, we will show how it may give a coarse-grained understanding of the system's performance characteristics.

3.1 Initial Performance Model

Figure 6 clearly shows that the BPEL process is composed of four distinct components with sequential behaviour. The first component, which describes the main flow of execution, is responsible for the reception of a purchase order and the final issue of an invoice. Between these two actions, three sequential

components perform some activities in parallel, as discussed above. To generate the performance model, it is necessary to associate each activity of the business process with a rate, which uniquely describes the exponentially distributed variable which indicates the duration of the activity. Listing 1.6 shows the PEPA sequential component corresponding to the main flow of execution. The operator prefix is used to describe the execution of an activity. Throughout this paper we adopt the convention that the activity name is the initials of the associated process name in Figure 6, and its rate of execution is indicated by `r_<name>`. (For example, the rate of *Receive Purchase Order* is `r_rpo`). The sequential component is cyclic so as to model the behaviour that the system is capable of processing a new order after the previous one has completed. The sequential components involved in price calculation, shipping management and production scheduling may be derived in a similar fashion. Their underlying performance models are shown in Listings 1.7, 1.8, and 1.9, respectively.

Listing 1.6. PEPA model of the main flow of execution of the BPEL process in Figure 6

```
ReceivePurchaseOrder = (rpo, r_rpo).InvoiceProcessing;
InvoiceProcessing = (ip, r_ip).ReceivePurchaseOrder;
```

Listing 1.7. PEPA model for price calculation

```
InitiatePriceCalculation = (ipc, r_ipc).CompletePriceCalculation;
CompletePriceCalculation = (cpc, r_cpc).InitiatePriceCalculation;
```

Listing 1.8. PEPA model for shipping management

```
DecideOnShipper = (dos, r_dos).ArrangeLogistics;
ArrangeLogistics = (al, r_al).DecideOnShipper;
```

Listing 1.9. PEPA model for production scheduling

```
InitiateProductionScheduling = (ips, r_ips).CompleteProductionScheduling;
CompleteProductionScheduling = (cps, r_cps).InitiateProductionScheduling;
```

In order to capture the business logic of the orchestration, these sequential components need to be augmented with further behaviour. Price calculation, shipping management, and production scheduling can only start after an order is received. Moreover, these activities may run in parallel. In SRMC this can be modelled by preceding their descriptions with a local state which synchronises over some action `fork`. Similarly, the invoice processing activity can be executed

after all the parallel flows are completed. To express this, a local state is added to the descriptions, which synchronises over the action `ip`. Thus, Listings 1.7, 1.8, and 1.9 are revised as shown in Listings 1.10, 1.11, and 1.12, respectively. In all descriptions, the names of the synchronising components are prefixed by the words `Fork` and `Join` and their activities are executed passively.

Listing 1.10. Revised PEPA model for price calculation

```
ForkPriceCalculation = (fork, _).InitiatePriceCalculation;
InitiatePriceCalculation = (ipc, r_ipc).CompletePriceCalculation;
CompletePriceCalculation = (cpc, r_cpc).JoinPriceCalculation;
JoinPriceCalculation = (ip, _).ForkPriceCalculation;
```

Listing 1.11. Revised PEPA model for shipping management

```
ForkShipper = (fork, _).DecideOnShipper;
DecideOnShipper = (dos, r_dos).ArrangeLogistics;
ArrangeLogistics = (al, r_al).JoinShipper;
JoinShipper = (ip, _).ForkShipper;
```

Listing 1.12. Revised PEPA model for production scheduling

```
ForkProductionScheduling = (fork, _).InitiateProductionScheduling;
InitiateProductionScheduling = (ips, r_ips).CompleteProductionScheduling;
CompleteProductionScheduling = (cps, r_cps).JoinProductionScheduling;
JoinProductionScheduling = (ip, _).ForkProductionScheduling;
```

Other sequential components are added to the system to observe the causality rules of the orchestration:

1. The `fork` action must be performed after `rpo` is executed.
2. The `cpc` action must be performed after `dos` is executed.
3. The `cps` action must be performed after `al` is executed.

Each rule is implemented by a cyclic two-state sequential component which observes the related actions in the order in which they must be executed. The component in Listing 1.13 models the first rule. An excerpt of the system equation (which will be fully shown later in this section) is presented in Listing 1.14. After a purchase order is received the component `Fork1` behaves as `Fork2`. This will in turn enable the activity `fork`, which will start the price calculation process.

Listing 1.13. Implementation of the causality rule for the execution of the parallel flows

```
Fork1 = (rpo, _).Fork2;
Fork2 = (fork, r_fork).Fork1;
```

Listing 1.14. Excerpt of the model's system equation showing the cooperation between the main flow and the price calculation component

```
(ReceivePurchaseOrder <rpo> Fork1) <fork> ForkPriceCalculation
```

Rules 2 and 3 are handled in a similar way: A cyclic two-state sequential component enforces the order of execution of the activities by enabling them passively. The corresponding sequential components used in this example are shown in Listings 1.15 and 1.16. Finally, the complete system equation for this model is shown in Listing 1.17.

Listing 1.15. Implementation of the causality rule between price calculation and shipping management

```
PriceShipping1 = (dos, _).PriceShipping2;
PriceShipping2 = (cpc, _).PriceShipping1;
```

Listing 1.16. Implementation of the causality rule between shipping management and production scheduling

```
ShippingProduction1 = (al, _).ShippingProduction2;
ShippingProduction2 = (cps, _).ShippingProduction1;
```

Listing 1.17. Complete system equation of the PEPA model

```
Sys = (ReceivePurchaseOrder <rpo> Fork1)
            <fork,ip>
              (
                (ForkPriceCalculation <cpc> PriceShipping1)
                   <fork,dos,ip> ForkShipper
              )
            <fork,al,ip>
              (ForkProductionScheduling <cps> ShippingProduction1)
```

Model of user workload. The system equation is combined with the model of user workload, which represents the external agents that invoke the service. Although various kinds of workload can be modelled, in this tutorial we shall consider the case of *closed workload*, i.e., a collection of users which cyclically execute the orchestration, interposing some think time between successive requests. The model of a user is shown in Listing 1.18. It describes a typical request/response scenario in which the activity `rpo` triggers the execution of the orchestration and the activity `ip` indicates the response of the system. The composition with the model of the orchestration is shown in Listing 1.19. Here, we made use of the array operator `[N_U]` to indicate N_U distinct users of the system.

Listing 1.18. Model of a user of a closed workload

```
Think = (think, r_think).Execute;
Execute = (rpo, r_rpo).Wait;
Wait = (ip, _).Think;
```

Listing 1.19. Complete model with user workload

```
Model = Think[N_U] <rpo,ip> Sys
```

System concurrency level. Similarly to the user workload, the concurrency levels of the system should be specified. The model in Listing 1.17 features one copy for each sequential component of the system. Thus, if the number of users is greater than one, then there is contention amongst these users to access the orchestration. After a user executes the shared action `rpo`, all the remaining users are *blocked* because the action cannot be enabled by the system. Indeed, the sequential component `ReceivePurchaseOrder` will behave as `InvoiceProcessing`, hence `rpo` is enabled again only after the current request has been completely processed.

In this scenario, one sequential component can be thought of as a single flow of execution which can handle only one request at all times. If multiple requests are to be handled simultaneously, then multiple copies of the sequential components need to be deployed. Again, SRMC makes use of the array operator to model this situation. The concurrency level has a clear impact on the performance of the system — it increases the system's throughput, or equivalently, reduces the response time. The concurrency level has also a clear counterpart in the actual system that the SRMC description is modelling. For instance it may refer to the number of threads or processes which are allocated to a given web service.

However, the modelling approach adopted in this case study poses an additional difficulty: although some sequential components clearly correspond to real units of execution, others have been introduced only to serve the auxiliary purpose of guaranteeing the intended order of execution of the business logic.

For this reason, the deployment of the latter kind of components will be dependent upon the concurrency levels of the sequential components which they are supporting.

One such example is the sequential component in Listing 1.15. To determine its concurrency level, we first observe that its initial local state `PriceShipping1` enables an action (i.e., `dos`) which must be executed in cooperation with the component `DecideOnShipper`, a local state of Listing 1.11. Let N_{FS} be the concurrency level of `ForkShipper` and N_{PS} be the concurrency level of `PriceShipping1`. If $N_{FS} > N_{PS}$ a reachable state of the system may have N_{FS} components in state `DecideOnShipper` and N_{PS} components in state `PriceShipping1`. Therefore, $N_{FS} - N_{PS}$ `DecideOnShipper` components cannot engage in the `dos` action. This introduces a form of blocking in the SRMC model which does not correspond to the real behaviour of the system, because `dos` is in fact an independent activity. Thus, it must hold that $N_{PS} = N_{FS}$ in order to avoid this undesired delay. On the other hand, the local state `PriceShipping2` enables a shared action which is carried out in cooperation with `CompletePriceCalculation`, a local state of the component in Listing 1.10. If the concurrency level of this component, denoted by N_{PC}, is lower than N_{PS}, then the activity `cpc` will be subject to delay. Conversely, if the concurrency level is higher, then some of the flows of `ForkPriceCalculation` will be under-utilised because there are not be enough requests to be served. In either case, the behaviour reflects that of the real system, thus the concurrency level of `ForkPriceCalculation` does not affect the calculation of the concurrency level of the auxiliary component `PriceShipping1`.

It is worthwhile pointing out that setting $N_{PS} > N_{FS}$ does not alter the performance results of the system. To understand this, observe that with such a setting there are more auxiliary components than the number of flows which can enable the `dos` action. The $N_{PS} - N_{FS}$ surplus components will be idle across the entire state space of the system. This is confirmed by the result that the state spaces of the models with $N_{PS} \geq N_{FS}$ are *lumpably equivalent* [1,6], which guarantees the equivalence of the derived performance measures.

Listing 1.20. Revised system equation of the PEPA model with concurrency levels

```
Sys = (ReceivePurchaseOrder[N_RP] <rpo> Fork1[N_RP])
       <fork,ip>
      (
       (ForkPriceCalculation[N_PC] <cpc> PriceShipping1[N_FS])
         <fork,dos,ip> ForkShipper[N_FS]
      )
       <fork,al,ip>
      (
           ForkProductionScheduling[N_PS]
           <cps>
            ShippingProduction1[N_FS]
        )
```

The concurrency levels of `ShippingProduction1` and `Fork1` can be determined with similar arguments. The revised system equation is shown in Listing 1.20.

3.2 Uncertainty about Parameters

The model described in the previous section may be subjected to quantitative analysis to extract performance indices. By interpreting it against the operational semantics of PEPA, a CTMC is built and transient as well as steady-state measures can be computed. However, one single analysis run often provides only a partial understanding of the system under study. More often, one is interested in the sensitivity of the performance to the variation of some of the system's parameters. For instance, one would like to ask questions such as: How is the system's performance affected by an increase in concurrency levels or an increase in the rate of execution of an action?

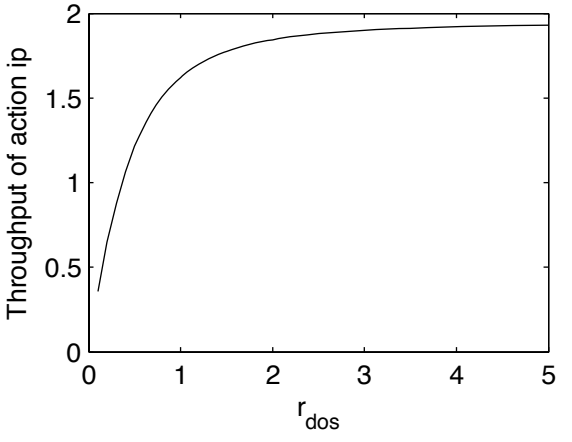
These questions arise at all stages of a modelling study. For example, performance analysis serves as a useful predictive tool which helps size the initial capacity of the system; further along the system's lifetime, it constitutes valuable support for planning system upgrades. With PEPA, answering these questions requires building a family of similar models, which maintain the same syntactic structure expect for one of more parameters which change in value. For instance, *sensitivity analysis* of the concurrency levels requires the construction of different models as in Listing 1.20 with distinct values for the parameters `N_RP`, `N_PC`, `N_FS`, or `N_PS`. If carried out manually, this process would be tedious and error prone. Fortunately, software tools for PEPA automate this form of analysis, requiring minimal user intervention [7,8].

If, on the one hand, this process is transparent to language, on the other hand the information about parameter uncertainty must be stored separately from the model. SRMC supports the declaration of *array of parameters*, which takes the form `param = { 1.0, 2.0, ... }`. This gives rise to distinct performance models in which the parameter `param` takes the different values in the set. (The syntax is not limited to numerical literals, but arbitrary expressions are also allowed.) This construct can be used for rate as well as concurrency level uncertainty. In the latter case, the elements of the set are restricted to be positive integers and expressions thereof.

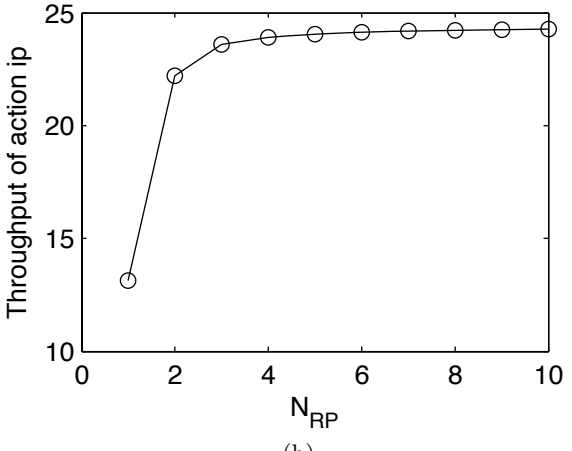
Figure 7 shows the results of two sensitivity analysis studies conducted on the model in Listing 1.20. In both cases the performance metric of interest is the system's steady-state throughput, indicated by the throughput of the action `ip`. Figure 7(a) studies its sensitivity with respect to the rate `r_dos`, specified in the SRMC model with the definition

```
r_dos = {0.1,0.2,...,5.0}.
```

These results enable the insight that the system benefits from increases in the rate of the `dos` activity, albeit the relative gain diminishes significantly in the region $[2.0, 5.0]$. For instance, doubling the rate from 0.5 to 1.0 gives a system's throughput improvement of about 30% while doubling the rate again from 2.0 to



(a)



(b)

Fig. 7. Examples of parameter sweep for the model in Listing 1.20. Sensitivity analyses are conducted with (a) rate `r_dos` and (b) concurrency level `N_RP`. In (a), `N_RP = 4` and in (b) `r_dos = 5`. In both cases the other parameters were set as follows: `N_U = 4`,`N_PC = 3`, `N_FS = 3`, `N_PS = 2`, `r_think = 1.0`; all other rates were set to 10.0.

4.0 provides only an improvement of about 4%. A similar qualitative behaviour is shown in Figure 7(b), which studies the sensitivity analysis with respect to the concurrency level of `ReceivePurchaseOrder`, by using the definition

```
N_RP = {1,2,...,10}.
```

Unlike the former, the latter analysis gives rise to structurally different underlying CTMCs because the values of `N_RP` alter the number of sequential components in the model and therefore the state space size.

3.3 Uncertainty about System Configurations

A further dimension of uncertainty is represented by dynamic binding. This is particularly interesting in service-oriented architectures, in which different providers may be functionally equivalent, i.e. they expose the same interface of the service. In these applications the identity of the services invoked by a client does not need to be known in advance, rather their binding is usually mediated by registries [9]. It is therefore a desirable feature to capture this form of dynamic binding in the SRMC model — although the services expose the same interface, their performance behaviour may vary significantly.

Listing 1.21. SRMC model for shipping management

```
r_rpo   = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0};
r_think = {1, 2, 3, 4};
r_ip    = 10.0; r_fork = 10.0; r_ipc = 10.0;
r_cpc   = 10.0; r_dos  = 5.0;  r_al  = 10.0;
r_ips   = 10.0; r_cps  = 10.0;

ForkShipper = SM::ForkShipper;
SM ::= { SM_A, SM_B, SM_C };

SM_A::{
  ForkShipper      = (fork, _).DecideOnShipper;
  DecideOnShipper  = (dos, r_dos).ArrangeLogistics;
  ArrangeLogistics = (al, r_al).JoinShipper;
  JoinShipper      = (ip, _).ForkShipper;
};

SM_B::{
  r_delay         = 5;
  ForkShipper     = (fork, _).DecideOnShipper;
  DecideOnShipper = (dos, r_dos).Delay;
  Delay           = (delay, r_delay).ArrangeLogistics;
  ArrangeLogistics = (al, r_al).JoinShipper;
  JoinShipper = (ip, _).ForkShipper;
};

SM_C::{
  r_fail           = 5;
  r_repair         = {1, 2};
  ForkShipper      = (fork, _).DecideOnShipper;
  DecideOnShipper  = (dos, r_dos).ArrangeLogistics
                   + (failure, r_fail).Failure;
  Failure          = (repair, r_repair).DecideOnShipper;
  ArrangeLogistics = (al, r_al).JoinShipper;
  JoinShipper      = (ip, _).ForkShipper;
};
```

Our language incorporates dynamic binding by means of *namespace* definitions. A namespace is used to isolate the definitions that are needed to fully define an individual component of the system under study. Functional equivalence among namespaces is then imposed through an array operator syntactically and semantically similar to the rate array operator introduced in Section 3.2. These features are now introduced by means of our running example. Let us suppose that the shipping management activities are outsourced and that they are accessed via web-service invocations. An interesting matter is to determine the impact on the overall orchestration of the shipping management services which can be bound. In order to answer this question, let us suppose that the dynamic behaviour of three of these services is known to the performance modeller. Each implementation of this service is assigned a namespace, `SM_A`, `SM_B`, and `SM_C`. The model of shipping management in Listing 1.11 is revised as shown in Listing 1.21. The namespace array `SM` indicates the set of possible bindings. Each element is a process definition within a namespace, accessed using the format `<namespace>::<process>`. Process and rate definitions are uniquely identified by their name as well as the namespace in which they are defined, thus the same definition can appear within distinct namespaces. This property has been exploited in our case study in order to stress the functional uniformity across the possible bindings. The implementation `SM_A` encapsulates the original definition of the process; `SM_B` interposes some external delay between the actions `dos` and `al`; finally, `SM_C` models a less reliable service which may fail. After failure occurs, some delay is introduced to reset the server to a fully working state. Additionally, `SM_C` has a rate array which indicates uncertainty about the rate of failure `r_f`.

This SRMC model gives rise to four underlying PEPA models, organised in a tree as shown in Figure 8. Each node represents a binding to a specific namespace or the selection of a rate of an array. Here, the nodes of the first level denote the binding to the three implementations of the shipping management component. The leaves indicate the parameter sweep across the rate of failure `r_f`. This

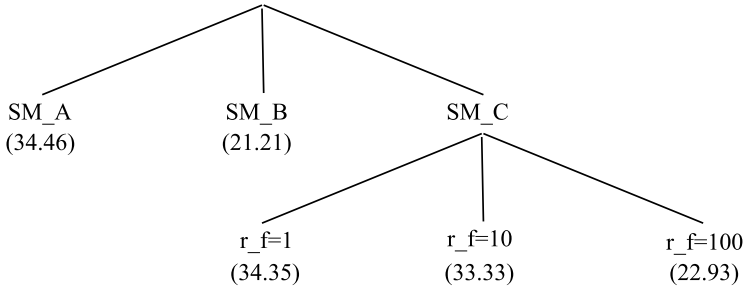


Fig. 8. The four PEPA models which underlie the SRMC description in Listing 1.21, represented as leaves of a tree. The nodes along the path from the root indicate the bindings to namespaces and the choice of rates for each PEPA model. In parentheses is the steady-state system throughput for each configuration.

analysis is not performed with `SM_A` and `SM_B` because these namespaces do not define a rate array. Each leaf is also labelled with the steady-state system throughput (action `ip`, in parentheses). The results show that the extra delay in `SM_B` has a negative effect on the performance. Conversely, the throughput does not deteriorate significantly when the rate of failure is relatively small, i.e. `r_f = {1,10}`. On the other hand, the perceived performance for `r_f = 100` is comparable to that of the system which binds to `SM_B`.

4 Deriving Experiments

In this section we explain precisely how all individual experiments are derived from a single SRMC model file. We will continue to use our running example for illustration. The aim is to derive a number of separate PEPA models corresponding to the number of distinct system configurations. We then analyse each PEPA model once for each configuration of the appropriate variable rates.

4.1 Namespace Scoping

The first part of our algorithm is to scope the identifiers used within namespaces. This is a straightforward translation which makes sure that each defined identifier is prefixed with the list of namespaces in which it occurs. There may be more than one if it occurs within a nested namespace. Additionally each reference to an identifier must be similarly prefixed in the same way as the definition was scoped. Shown below is the scoping as applied to the `SM_B` namespace:

Listing 1.22. Scoped `SM_B`

```
SM_B::{
  SM_B::r_delay          = 5;

  SM_B::ForkShipper      = (fork, _).SM_B::DecideOnShipper;
  SM_B::DecideOnShipper  = (dos, r_dos).SM_B::Delay;
  SM_B::Delay            = (delay, SM_B::r_delay).SM_B::ArrangeLogistics;
  SM_B::ArrangeLogistics = (al, SM_B::r_al).SM_B::JoinShipper;
  SM_B::JoinShipper      = (ip, _).SM_B::ForkShipper;
};
```

Notice that action names are not scoped, this allows cooperation in the final system equation between components defined in separate namespaces. The globally defined rate `r_dos` is also not prefixed with any namespace since this rate is defined at the top level.

4.2 Namespace Selections

The key component in distinguishing system configurations in SRMC models is the namespace selection definitions. In our example this is the line:

```
SM ::= { SM_A, SM_B, SM_C };
```

We do a depth first search of all namespace selections because some namespace selections will entail further namespace choices. This occurs when we have a namespace selection nested within a namespace which is itself a choice. In our example this does not occur and in fact we have only one choice leading to three system configurations. Once a namespace has been chosen there are two tasks left to do; the first is to promote the definitions made within the given namespace into the top level and remove the now empty namespace definition. The second task is to substitute references to the selection definition (in this case `SM`) for references to the selected namespace. In this example we have been quite frugal in our use of the abstract namespace `SM`, so there is only one line to change other than the promoted definitions, namely the line:

```
ForkShipper = SM::ForkShipper;
```

Because we made this definition and then used the name `ForkShipper` the system equation is the same for all three models. However in general the system equation is modified according to the namespace selections. So that references to `SM::ForkShipper` and `SM::DecideOnShipper` would be replaced with references to `SM_A::ForkShipper` and `SM_A::DecideOnShipper` and respectively so for the other definitions and the other derived models.

The completion of these two tasks results in what is an almost valid PEPA model for each possible configuration. The remaining non-valid PEPA definitions are those rate definitions with uncertainty; these are removed in the next section. The (almost valid) PEPA model given in Listing 1.23 corresponds to choosing `SM_A` and the Listings 1.24 and 1.25 show the differences between that model and the derived instances resulting from choosing `SM_B` and `SM_C` respectively.

Listing 1.23. Scoped PEPA model example

```
r_rpo   = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0};
r_think = {1, 2, 3, 4};
r_ip    = 10.0; r_fork = 10.0; r_ipc = 10.0;
r_cpc   = 10.0; r_dos  = 5.0;  r_al  = 10.0;
r_ips   = 10.0; r_cps  = 10.0;
```

```
ReceivePurchaseOrder = (rpo, r_rpo).InvoiceProcessing;
InvoiceProcessing = (ip, r_ip).ReceivePurchaseOrder;

Fork1 = (rpo, _).Fork2;
Fork2 = (fork, r_fork).Fork1;

ForkPriceCalculation = (fork, _).InitiatePriceCalculation;
InitiatePriceCalculation = (ipc, r_ipc).CompletePriceCalculation;
CompletePriceCalculation = (cpc, r_cpc).JoinPriceCalculation;
JoinPriceCalculation = (ip, _).ForkPriceCalculation;

PriceShipping1 = (dos, _).PriceShipping2;
PriceShipping2 = (cpc, _).PriceShipping1;

ForkShipper = SM_A::ForkShipper;
SM_A::ForkShipper      = (fork, _).SM_A::DecideOnShipper;
SM_A::DecideOnShipper  = (dos, r_dos).SM_A::ArrangeLogistics;
SM_A::ArrangeLogistics = (al, r_al).SM_A::JoinShipper;
SM_A::JoinShipper      = (ip, _).SM_A::ForkShipper;
SM_A::Deciding         = [ SM_A::DecideOnShipper ];

ShippingProduction1 = (al, _).ShippingProduction2;
ShippingProduction2 = (cpc, _).ShippingProduction1;

ForkProductionScheduling = (fork, _).InitiateScheduling;
InitiateScheduling = (ips, r_ips).CompleteProduction;
CompleteProduction = (cps, r_cps).JoinProductionScheduling;
JoinProductionScheduling = (ip, _).ForkProductionScheduling;

User    = (think, r_think).Execute;
Execute = (rpo, _).Wait;
Wait    = (ip, _) .User;


Waiting = [ Wait ];

n_u   = {7000, 8000, 9000};
n_rp  = 800;
n_pc  = {500, 600, 700};
n_fs  = {500, 600, 700};
n_ps  = 400;

User[n_u] <rpo, ip>
  (  (ReceivePurchaseOrder[n_rp] <rpo> Fork1[n_rp])
      <ip,fork>
     (  (ForkPriceCalculation[n_pc] <cpc> PriceShipping1[n_fs])
         <dos,fork,ip>
         ForkShipper[n_fs]
     )
```

```
      <fork,al,ip>
     (  ForkProductionScheduling[n_ps] <cps>
        ShippingProduction1[n_fs]
     )
  )
```

Listing 1.24. Scoped PEPA model example

```
// Same as in the other model

ForkShipper = SM_B::ForkShipper;
SM_B::r_delay          = 20;

SM_B::ForkShipper      = (fork, _).SM_B::DecideOnShipper;
SM_B::DecideOnShipper  = (dos, r_dos).SM_B::Delay;
SM_B::Delay            = (delay, SM_B::r_delay).SM_B::ArrangeLogistics;
SM_B::ArrangeLogistics = (al, SM_B::r_al).SM_B::JoinShipper;
SM_B::JoinShipper      = (ip, _).SM_B::ForkShipper;
SM_B::Deciding         = [ SM_B::DecideOnShipper ];
// The rest is also the same including the system equation
```

Listing 1.25. Scoped PEPA model example

```
// Same as in the other model

ForkShipper = SM_C::ForkShipper;
SM_C::r_fail = 5;
SM_C::r_repair = {1, 2};
SM_C::ForkShipper     = (fork, _).SM_C::DecideOnShipper;
SM_C::DecideOnShipper = (dos, r_dos).SM_C::ArrangeLogistics
                      + (failure, SM_C::r_fail).SM_C::Failure;
SM_C::Failure = (repair, SM_C::r_repair).SM_C::DecideOnShipper;
SM_C::ArrangeLogistics = (al, r_al).SM_C::JoinShipper;
SM_C::JoinShipper = (ip, _).SM_C::ForkShipper;
SM_C::Deciding = [ SM_C::DecideOnShipper + SM_C::Failure ];

// The rest is also the same including the system equation
```

4.3 Rate Parameter Experiments

The rate parameter selections are removed simply by creating a standard PEPA rate specification using the first of the possible selections for the rate. This gives us three PEPA models which are written out to three files: `SM_A.pepa` and `SM_B.pepa` and `SM_C.pepa`. It remains only to perform sensitivity analysis

over these PEPA models. We create an experiment for each which ranges over all possible combinations of the rate selections, we use these to override the definitions given in the standard PEPA model using substitution. The first two PEPA models corresponding to the choices `SM_A` and `SM_B` use only globally defined rate choices while the choice of `SM_C` uses a rate choice which need not be ranged over for the other models. So the number of experiments will be larger for the configuration in which `SM_C` is chosen. Where the `...` stand for the rest of the arguments given to compute the specified measure, the experimentation for the first configuration begins with:

```
ipc --rate r_po=1,r_think=1,n_u=7000,n_pc=500,n_pc=500 ...
ipc --rate r_po=2,r_think=1,n_u=7000,n_pc=500,n_pc=500 ...

ipc --rate r_po=1,r_think=2,n_u=7000,n_pc=500,n_pc=500 ...
ipc --rate r_po=2,r_think=2,n_u=7000,n_pc=500,n_pc=500 ...

ipc --rate r_po=1,r_think=3,n_u=7000,n_pc=500,n_pc=500 ...
ipc --rate r_po=2,r_think=3,n_u=7000,n_pc=500,n_pc=500 ...
...
```

In this way we range over all possible rate configurations appropriate for the `SM_A` system configuration. We are not ranging over the rate `SM_C::r_repair` because this rate is not used in the first (or second) configuration. The experiment for the second configuration looks much the same, for the third we must range over the extra rate `SM_C::r_repair` and our experiment looks like:

```
ipc --rate r_po=1,r_think=1,n_u=7000,n_pc=500,n_pc=500,r_repair=1
ipc --rate r_po=2,r_think=1,n_u=7000,n_pc=500,n_pc=500,r_repair=1

ipc --rate r_po=1,r_think=2,n_u=7000,n_pc=500,n_pc=500,r_repair=1
ipc --rate r_po=2,r_think=2,n_u=7000,n_pc=500,n_pc=500,r_repair=1

ipc --rate r_po=1,r_think=3,n_u=7000,n_pc=500,n_pc=500,r_repair=1
ipc --rate r_po=2,r_think=3,n_u=7000,n_pc=500,n_pc=500,r_repair=1
...
```

The number of experiments produced for each of the derived PEPA models is equal to the product of the lengths of all the appropriate rate selections. For the `SM_A` PEPA model this is $6 \times 4 \times 3 \times 3 \times 3 = 648$ and the same is true for the second (`SM_B`) configuration. For the `SM_C` configuration it is this number multiplied by the extra uncertainty of the rate `r_repair` which is $2 \times 648 = 1296$ adding these all together we get the total number of experiments to be 2592.

5 Query Specification

When performing analysis over a SRMC model we generate many — perhaps several thousand — PEPA model instances which must all be analysed separately. Clearly we do not wish to analyse each of these PEPA model instances

by hand but automatically. We therefore require a query specification technique that is portable across many similar models. Our query specification language is that of eXtended Stochastic Probes [10]. We enhance this by allowing *virtual components*.

When specifying a measurement we are often concerned with specifying a set or sets of states. In the Markovian world these states are the states of the CTMC which underlies the PEPA model in question. Even when analysing a single PEPA model one does not wish to specify the states of the Markov chain directly since these are automatically derived from the PEPA model. We wish to specify such states compositionally just as we have compositionally described the model. One method of doing this is with a state specification where the full state space of the model is filtered with respect to the population sizes of the sequential states of the individual components. Figure 9 reports the grammar of state specifications.

expr	:=	*Process*	population
	|	*int*	constant
	|	*expr relop expr*	comparison
	|	*expr binop expr*	arithmetic
relop	:=	$= \mid \neq \mid > \mid <$	
	|	$\geq \mid \leq$	relational operators
binop	:=	$+ \mid - \mid \times \mid \div$	binary operators
pred	:=	$\neg$*pred*	not
	|	true | false	boolean
	|	if *pred*	
		then *pred*	
		else *pred*	conditional
	|	*pred* &&& *pred*	disjunction
	|	*pred* $\|\|$ *pred*	conjunction
	|	*expr*	expression

Fig. 9. The full grammar of the state specifications

State specifications can work well for measurements of steady-state condition probabilities but are not so appropriate for passage-time measurements. This is because for passage-time analysis we are concerned with events which happen and the states which result from those events. This means that slight changes to the model can affect the passage-time specification greatly because there are more or fewer states along the passage. In SRMC we mitigate this to some extent with our use of virtual components. A virtual component is one which has no representation within the CTMC but takes its population value as a function of the populations of other related components. For example using the following definition it is possible to define a component whose population is a measure of the number of components which are in either the `Broken` state or the `Offline` state:

Listing 1.26. Virtual Component for an unavailable service

```
Unavailable = [ Broken + Offline ] ;
```

When we wish to measure the states of a component that correspond to some abstract state, such as being unavailable or being within a passage to be measured, we can use a virtual component to ensure that the query specification may be the same across several configurations. When modelling with a single PEPA model this can be useful in that the states along the passage are defined in the same place as the behaviour of the component(s) involved in the passage — such as a user component which is in an abstract state of *Waiting* in order to analyse reponse-time. This is especially useful in SRMC when the definitions for a service component can change based on the system configuration in use for a specific derived PEPA model. In our running example we use virtual components to specify when the shipping component is in a state of deciding on a shipper. For configurations `SM_A` and `SM_B` this is simply one local state. For configuration `SM_C` though the shipping component may be in the `DecideOnShipper` state or the `Failure` state in which it is still in the (delayed) process of deciding on the shipper. So we make the virtual component with:

Listing 1.27. Virtual Component for deciding

```
Deciding = [ DecideOnShipper + Failure ];
```

For each of the configurations `SM_A` and `SM_B` this is simply:

Listing 1.28. Virtual Component for deciding

```
Deciding = [ DecideOnShipper ];
```

Now when we make the selection we can simply refer to `SM::Deciding` in this way we have a measure of the number of components in the abstract state of 'Deciding' which is portable across all of the derived PEPA models.

Activity probes can allow a more intuitive query specification when the states we are interested in are the results of a sequence of event observations. This is the common case when the query is a passage-time query. In `xsp` the modeller specifies a series of activities to be observed and the compiler automatically translates this into a PEPA sequential component which can then be queried as a filter on the entire state space of the model. For passage-time measurements the user can label activities of the probe as either `start` activities which begin the passage or `stop` activities which end the passage. The probe states which are

source, target or passage states are then mechanically derived and given to the analyser. The user therefore need not specify the states they are interested in at all, only the events/observations. A very common passage-time query which measures the response-time is given by the probe specification:

Listing 1.29. Response-time probe specification

```
request:start, response:stop
```

In our example the request is started with the completion of a `think` activity and is terminated with the completion of a `ip` activity. So the equivalent probe for our example model is:

Listing 1.30. Response-time probe specification

```
think:start, ip:stop
```

However often we are concerned with the response-time as observed by a single client, rather than that observed by the system above. The above probe will measure the passage between the occurrence of a `request` activity performed by any component in the model (usually a synchronisation between one client and the service being modelled) and an occurrence of a `response` activity again performed by any component. To observe only those `request` and `response` activities which originate from a single 'tagged' client we can attach the probe to a single `Client` component rather than the whole model. The following probe using the double colon syntax achieves this for our example:

Listing 1.31. Response-time probe specification

```
User::(think:start, ip:stop)
```

Sometimes events are not powerful enough to express the queries that we are interested in. This is often the case when we are interested in the response-time when the service is in a particular state. For example we may have made one passage-time measurement already using the above probe and found that the general response-time is adequate, but that we wish to know more about how this general response-time profile is made up. One possibility is to split up the query into several analysing response-times when the service is in different states. The following two probes analyse the response-time for all requests that are initially made when the service is entirely available or (at least) partially unavailable.

Listing 1.32. Response-time probe specification

```
User::({Unavailable == 0}think:start, ip:stop)
User::({Unavailable > 0}think:start, ip:stop)
```

The guards on the first activity observation (of the `think` activity) are state specifications. Note that these may refer to virtual components as well as to regular component populations. Guards need not always be used to make a distinction as to when a passage is begun, they may also be used to terminate the passage. The following two probes analyse the time it takes for a system to become fully repaired after the initial breakdown of one of its components/servers.

Listing 1.33. Response-time probe specification

```
break:start, {Broken == 1}repair:stop
break:start, {Broken == 0}:stop
```

Here we do not assume that there is a single server which may be broken or not, but several servers each of which may be broken independently. So the probe must begin the passage on observation of the first breakage and only terminate the passage when a `repair` activity fixes the only broken server (other servers may have broken since the first one did). The first probe achieves this by only observing a `repair` activity if there is exactly one server in the `Broken` state. However this may still not be robust enough since we may change the model such that a `repair` does not necessarily fix the broken server, for example there may

P_{def}	:=	$name :: R$	locally attached probe
	|	R	globally attached probe
R	:=	$activity$	observe action
	|	R_1, R_2	sequence
	|	$R_1 \mid R_2$	choice
	|	R:label	labelled
	|	$R/activity$	resetting
	|	(R)	bracketed
	|	$R\ n$	iterate
	|	$R\{m, n\}$	iterate
	|	R^+	one or more
	|	R^*	zero or more
	|	$R^?$	zero or one
R	:=	$\ldots \mid \{pred\}R$	guarded

Fig. 10. The full grammar of the eXtended Stochastic Probes query specification language

be more than one thing broken within the server. The second probe by contrast has the guard placed on the `:stop` label itself. This means that the probe will consider itself to have terminated the passage precisely when we first move into a state in which there are no servers in the `Broken` state. Figure 10 provides the full grammar for the `xsp` language.

6 Markovian Modelling with Many Models

In the Markovian world in which we generate a CTMC from a PEPA model we suffer from the well-known state-space explosion problem. Small increases in the size of the PEPA model or the population size of a component or components within the model can cause the number of distinct states of the generated CTMC to increase dramatically. We mitigate this to some extent with our use of aggregation [6] but this can provide only so much relief — this is described in [11]. When the model state space becomes large we cannot simply allow more time for the numerical solver because at some point the state space is simply too large to even generate or hold in memory. From our single SRMC model we generate a large number of PEPA models each of which can be solved independently. Because we do not model uncertainty by increasing the complexity of a single large PEPA model then provided each derived PEPA model is not itself too large we avoid state space explosion. In other words we can solve many small models better than we can solve one very large model. Indeed each of the generated PEPA models may be solved in parallel on many machines using a grid or cluster computing environment. The task of separately solving each of these models falls into the class of problems which are known in the parallel computing community as "embarrassingly parallelisable". That is to say, they are essentially a large number of independent processes without synchronisation points and therefore they deliver impressive speedups when executed on a compute cluster.

Even with this the model sizes which can be solved using the CTMC technology is still low. Although in our example model presented so far we have several thousand users and several hundred server processes. Unfortunately this would result in an unmanageably large CTMC with a state-space size described in astronomical terms. Given this, instead of analysing the whole system we are obliged to analyse a portion of it, such as the performance of one set of server processes. To this end we modified our SRMC model to have one for each kind of server process — this means we set the values N_{RP}, N_{PC}, N_{FS} and N_{PS} all to one. We are therefore able to reduce the number of clients since some clients will be served by other server processes. In this example we ranged over the number of users N_U with the SRMC definition:

Listing 1.34. Number of users specification

```
N_U = { 2, 3, 4, 5, 6};
```

This also meant that we are not ranging over the values N_{PC} and N_{FS} and hence there were not as many experiments to run. We ran 750 experiments each of which took between 30 seconds and one 1 minute to complete on an ordinary desktop computer.

Prior to solving each of the models we must specify some query with which to analyse each model. This is because some queries require us to automatically add components to the model in order to distinguish states. For our model of the web service we are interested in the response-time as observed by a single user. Often we are interested in average response-time but compiling the PEPA models to CTMCs allows a finer grained analysis known as passage-time quantile analysis [12,13]. This allows the prediction of not just the average response-time but the response-time profile, such that we know the probability of receiving a response at or within any given time t after the request was made. This allows us to answer such service-level agreements as: "90 percent of all requests will be serviced within 10 seconds" something which is not possible to answer with only the average response-time. Having specified this as our performance query once for the SRMC model, this is then translated into the equivalent query for each of the generated PEPA models. Thus, for each of the generated PEPA models we calculate a cumulative distribution function which plots the time t for a specified range (in this case 0 to 10) against the probability that a specific user observes the `ip` event t time units after performing a `think` action.

Having calculated this function for each of the generated PEPA models we now have a database mapping process instantions and rate parameters to response-time profiles. We can extract information from this database as we wish. The graph on the left of Figure 11 shows for one specific set of process instantiations (or system configuration) the response-time profile as we vary the number of users. All the other rates are held constant — by this we mean that we have selected results from runs which have the same parameters other than the number of users. This graph indicates that the number of users has a quite dramatic effect on the response-time of the system, where there is a low number of users the probability of passage completion rises very quickly with time. As the number of users is increased the rise of the probability of completion against time is more languid. The graph on the right hand side of the same figure does the same kind of analysis except here we have kept constant the number of users and the parameter that we are varying — whereby again 'varying' means selecting the already computed results which correspond to a varying — rate of `rpo` the rate at which the service can receive orders. From this graph we learn that at least for the parameter range chosen the rate of `rpo` does not drastically affect the response-time as observed by a single user. This is a perhaps surprising result because the activity performed at this rate is included within the analysed passage.

Figure 12 shows a similar kind of surface plot except in these graphs we are holding only the system configuration as constant and ranging over the whole rate configuration space for each derived PEPA model. What you see is the depths of probabilities at each time for the given system configuration. Where

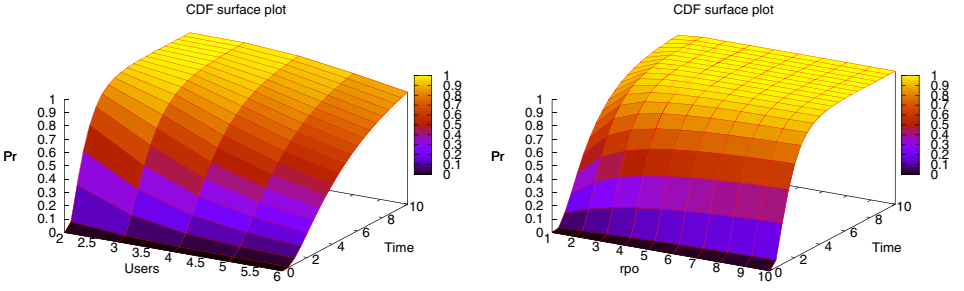


Fig. 11. A surface plot showing how the number of users affects the response-time profile

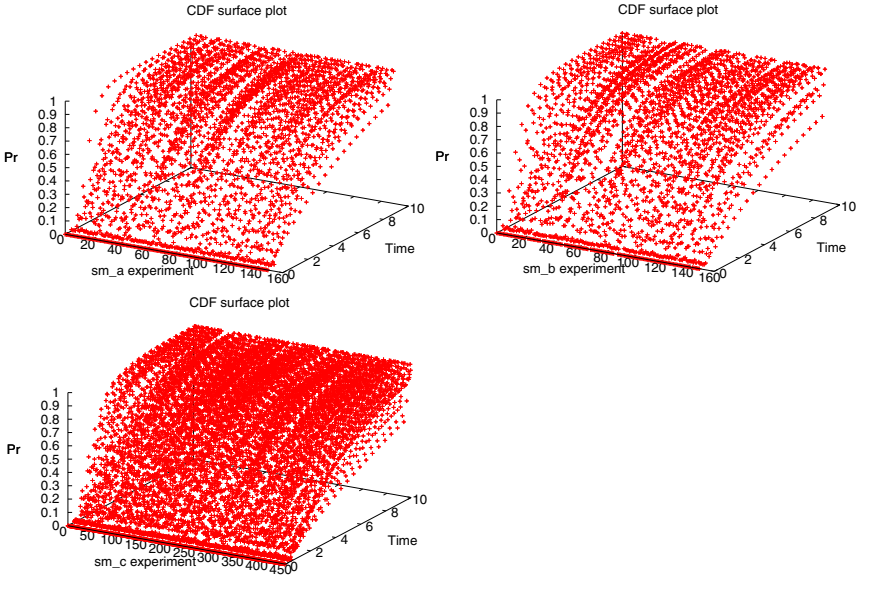


Fig. 12. Surface plots depicting the cumulative distribution functions for each system configuration across all rate configurations

this depth is long there is great variability in the probability of completing the passage at that time. In other words at that time the rates have a large affect on the probability of completing the passage. Where the depth is low the rates do not affect so much the probability of completion, this may be because there is either always low or always high proability at that time or it may be because there is some bottleneck in the passage and therefore altering other rates has less effect.

Figure 13 depicts the candle-stick graphs of completion of the passage for all the experiments performed automatically from the SRMC model. At each

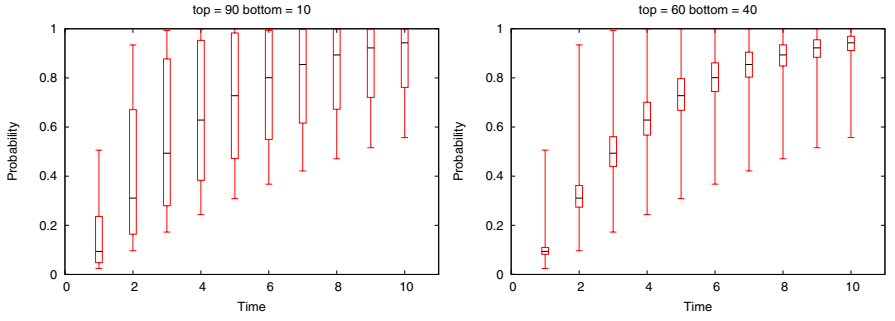


Fig. 13. Candlestick graphs showing the probability of completion ranging over all of the experiments performed automatically from the SRMC model

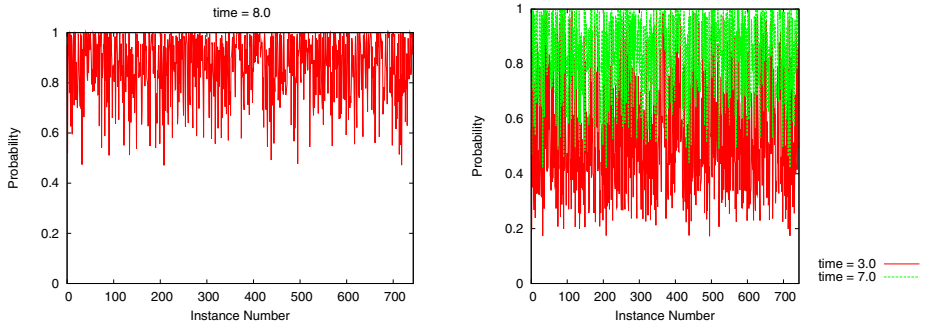


Fig. 14. Time graphs showing probability of completion at the given times plotted against experiment number

time point the top and bottom bars represent the best and worst performing experiment at that time point. We see from these two graphs that the range of possibilities is quite high which suggests that the exact system configuration is important for the modelled system. The thick bar along each line represents a particular middle-percentage range for that time — in other words we remove some percentage of best performing experiments and some percentage of the worst performing experiments. The graph on the left plots between 10 and 90 percent while the graph on the right plots between 40 and 60 percent. In general this can highlight the possibility that the best or worst performing experiment is really an outlier and the wide variability of the experiments is not a true reflection of the variability of the system as a whole. It can also allow the modeller to zero in on experiments which are causing particularly good or poor results and determine whether or not the system/model can be improved as a whole.

Another kind of graph which we plot are called simply *time graphs*. These plot the probability of completing the passage within the given times against all the experiment numbers of all the system and rate parameter configurations ranged over within the SRMC model. On the left of Figure 14 we see only the single time

8.0 being plotted. From this graph we can see that with few exceptions there is at least a sixty percent chance that the passage will be completed and in some configurations it is close to a certainty. On the graph of the right of the same figure we plot more than one time, indeed our software ranges over sets of times although more than two is not very useful when the graphs are in black and white. From these two times we can see that there are some configurations that are more likely to complete the passage within time 3.0 than other poorer performing configurations are at time 7.0. Again this demonstrates wide variability in the system we are modelling.

7 Large-Scale Modelling with Differential Equations

Our use of SRMC to model uncertainty by splitting up the possible system configurations has ensured that our model does not inflate to an unmanageable size through the modelling of uncertainty. However there are many models which are inherently large, in particular models of web services often hope to have many thousands of users. Therefore even when modelling one single configuration the state-space size is simply too massive. Recently it has become possible to analyse such systems with a fluid-flow approach. In this case the PEPA model is translated into a system of ordinary differential equations. These are solved until the model has reached a steady-state in which the population levels of each kind of component are stationary. This gives us the same kind of steady-state measurements that are possible with the CTMC analysis. Systems in which the limit of user components for CTMC analysis was of the order of a few tens can now be analysed with a more realistic number of users in the many thousands. Unfortunately the price paid for this extraordinary rise in model size capacity is a reduced set of analyses which are appropriate. In particular our passage-time analysis used on the CTMC models of the Section 6 in as of yet unavailable.

In our example model we can instead calculate a measure of the average response-time by looking at the number of users who are typically in a state of waiting for their response. For each derived PEPA model instance and rate configuration we solve the assocated ODEs to provide a time-series analysis. These plot the population of the specified component types against time. Three such graphs are shown in Figure 15 one for each of the three system configurations. Note that after some time each of these time-series becomes stable in that the population of each component type is not changing. This allows us to infer the steady-state or long-term average poplulation of each component type. By analysing the long-term population level of the number of waiting users we can gain a measure of the response-time of the system. The graph in Figure 16 plots the steady-state population of waiting users for all of the experiments (that is all system configurations at all rate configurations). We did the same for the number of deciding shippers (recall from Section 5 that 'Deciding' was a virtual component) and the results are plotted in Figure 17. Overall we ran 2592 experiments each of which took between 1 and 3 seconds to complete. We invoked these in serial on an ordinary desktop PC and achieved results within 2 hours.

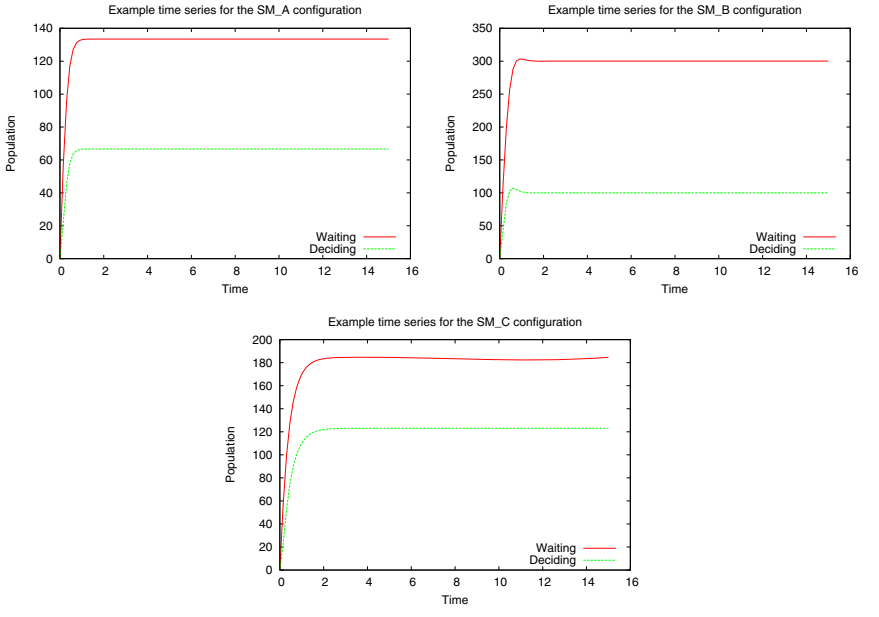


Fig. 15. Example time-series showing how the population of a subset of the component types in specific model instances change with time

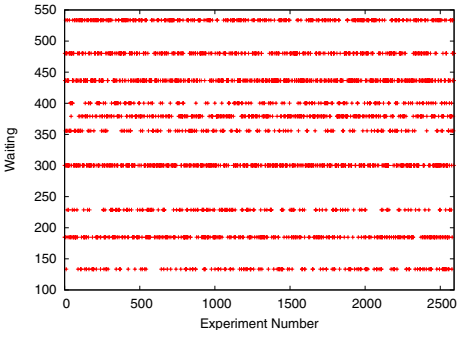


Fig. 16. Graphs showing how configurations affect the number of waiting users

For some models the time taken to produce one result is longer, alternatively we may have a larger uncertainty space resulting in many more experiments. In these cases it is worth considering farming out the solving of each experiment (or a set of experiments) using a parallel computing cluster such as Condor as we have done before [14].

The results show that the population levels tend to concentrate on a very small number of values, as can be intuitively appreciated by the presence of horizontal lines in the graphs. An explanation of this behaviour is that there are dominant elements in the parameter space considered in this case study. Particularly, the rate `r_rpo` and the instance of `SM` have a strong impact on

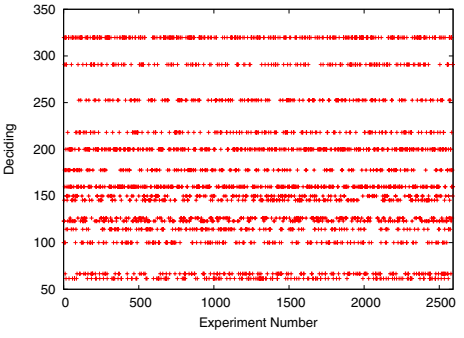


Fig. 17. Graphs showing how configurations affect the number of deciding shippers

Table 1. Results of Figure 16 grouped by `SM` and `r_rpo`. The third column shows the average population level across all the experiments with the same configuration.

SM	r_rpo	Average Value
SM_A	0.5	66.65
SM_A	1.0	114.25
SM_A	1.5	149.94
SM_A	2.0	177.70
SM_A	2.5	199.90
SM_A	3.0	218.07
SM_B	0.5	61.52
SM_B	1.0	99.97
SM_B	1.5	126.28
SM_B	2.0	145.40
SM_B	2.5	159.85
SM_B	3.0	159.85
SM_C	0.5	123.05
SM_C	1.0	199.95
SM_C	1.5	252.56
SM_C	2.0	290.91
SM_C	2.5	319.69
SM_C	3.0	319.69

the population levels considered. Table 1 gives the results of Figure 16 in an aggregated form by grouping the experiments according to these two parameters. It shows the average population level across all the experiments with the same values of `r_rpo` and `SM`. Each group of experiments exhibits a negligible standard deviation, confirming the strong influence of the two parameters on the performance measure.

The specific instance of `SM` seems to have a stronger impact in some cases. For instance, the groups of experiments (`SM_B`, 2.5) and (`SM_B`, 3.0) have the same average population level, suggesting a low sensitivity of result with respect to the change in the value of the rate `r_rpo`. The same situation is observer for the groups (`SM_C`, 2.5) and (`SM_B`, 3.0).

8 Software Tools for SRMC and PEPA

The kinds of analysis presented in this tutorial are supported by a suite of software tools which can be accessed at `http://groups.inf.ed.ac.uk/srmc/`. The tool smc implements the compiler for SRMC, and is based on Pepato and ipc for the quantitative analysis. This section gives an overview of the tool-chain and discusses the main features of each individual application. The architecture of the software for SRMC is depicted in Fig. 18.

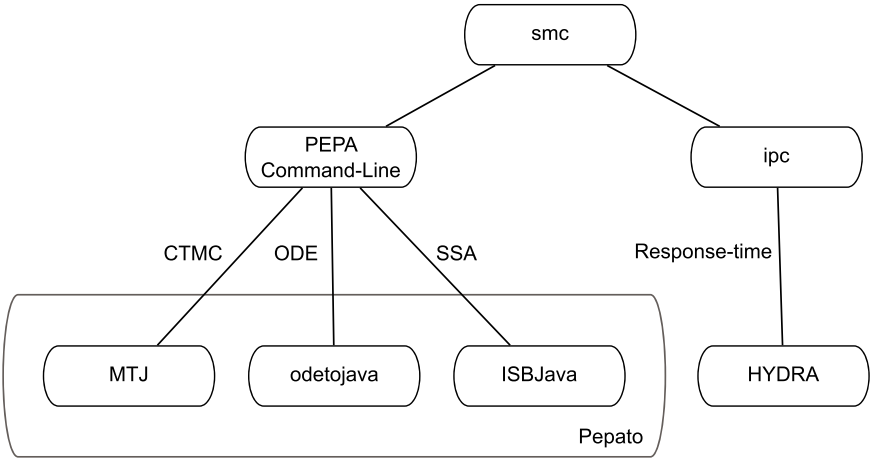


Fig. 18. Architecture of the software tools for SRMC

8.1 Pepato

Pepato is a Java Application Programming Interface (API) which provides core services for the execution of PEPA-related tasks. The API is centred around an abstract syntax tree representation of a PEPA model which can be created programmatically or via the parsing of a file with the concrete syntax presented throughout this tutorial. This in-memory representation gives access to functionality for Markovian analysis and fluid-flow approximation through ODEs.

The modules for Markovian analysis include a *state-space explorer*, which infers the labelled transition system according to the semantics rules of PEPA. The generated state space is the input to the *steady-state analyser* which constructs the underlying Markov chain and solves it for the equilibrium distribution using the external library MTJ (Matrix Toolkit for Java) [15]. The solution can be used for the calculation of a predefined set of performance indices, indicated as reward structures over the Markov chain: the *throughput* is calculated for each action type, and the mean population level is given for each sequential process in the system. An alternative form of Markovian analysis is *stochastic simulation*, which is offered by Pepato with the implementation of a semantics for PEPA [16] which

maps onto efficient methods such as Gillespie's direct method [17] and Gibson-Bruck's algorithms [18]. This module, which relies on the stochastic simulation algorithms implemented by the Java library ISBjava [19], permits the tuning of the most common parameters of a simulation study, including number of replications, time horizon, confidence intervals, and output variables of interest.

The module for fluid-flow analysis transforms a PEPA model into a system of first-order ODEs according to the semantics described in [20]. The differential equation model can be analysed with a range of numerical solvers and the output provides the evolution of the population levels of the system's components over time. The module uses the odetojava package by Patterson and Spiteri [21], available in the ISBjava library.

Pepato is exposed through a command-line interface for the purpose of communication with the other non-Java elements of the SRMC tool-chain.

8.2 ipc

The ipc tool is written in Haskell and supports the analysis of response-time quantiles for PEPA [13]. The tool permits the definition of performance measures (called *probes*) by using a regular-expression like specification language called xsp which we described in Section 5. The tool converts a PEPA model and its measurement specification into an equivalent PEPA model for the numerical analysis. In its latest incarnation, backed by the library ipclib [7], the solvers are implemented natively. Optionally, the user can invoke the original tool-chain, which translates the model into the input format accepted by Hydra [22], implemented in C++, which has been designed to cope efficiently with large-sized models (i.e. up to 10^7 states).

8.3 smc

The SRMC Model Compiler (smc) is our software support for translating from the SRMC language into multiple PEPA models. The tool is written in Haskell alongside the ipc library and hence uses the same parsing for the PEPA specific portions of the SRMC syntax. The user provides both a SRMC model together with a query specification. From this, one PEPA model per system configuration is produced. The query specification is robust enough to be used over all the PEPA models. Note that we do not produce one PEPA model for each experiment. Each experiment consists of a system configuration (or equivalently a derived PEPA model) and a set of rate parameter instantiations. So our smc compiler produces for each derived PEPA model a list of rate parameter instantiations relevant to that particular model instance. For each experiment the ipc tool is invoked with the particular PEPA model, the particular rate parameter instantiations and the globally appropriate query specification. The ipc tool may then produce the result itself for a Markovian response-time quantile query or pass the instantiated model onto Pepato to solve using ordinary differential equations.

9 Related Work

This tutorial paper is concerned with using process calculi to model Web Services with a particular focus on quantitative evaluation. As others have before us, we used a process calculus to express our model. The process calculus SRMC builds on the simpler process algebra PEPA which has been used for numerous studies of stochastically-timed systems (for a recent overview of modelling with PEPA see [23] and [24]).

SRMC can be seen as an extension of PEPA. Formally, it is a superset of the PEPA language in the sense that every PEPA model is immediately an SRMC model which gives rise to a singleton set of PEPA model instances. Another extension of PEPA which had *mobility* (rather than *binding*) as its motivation was the language of PEPA nets [25,26]. PEPA nets are stochastic Petri nets whose tokens are PEPA terms. We find this language applied to modelling Web Services in [27].

PEPA nets and the stochastic π-calculus are applied to modelling a Web Service in [28]. Of note with regard to this paper is that properties of interest to the PEPA net model are specified using PML$_\nu$, an extension of Larsen and Skou's Probabilistic Modal Logic (PML). This is a distinctive approach to characterising sets of states which are of interest in the specification of performance measures. Modal logics are rarely used for this purpose and temporal logics such as CSL are more commonly applied here.

However, the style of modelling pursued in the above work is discrete-state Markovian modelling. With the advent of the theory presented in [2] and the algorithm presented in [29] it was possible to map process algebra models to systems of ordinary differential equations for solution. The relationship between these two kinds of models is explored from a theoretical perspective in [30] and by example in [31] and [11].

One of the earliest published papers to include a PEPA modelling case study which is carried out using this evaluation method is [32]. In this paper a large-scale model of the BitTorrent distribution protocol is developed in PEPA and solved using the fluid-flow approximation. Previous models of Web Services considered in this style include the distributed e-learning and course management system considered in [33] in PEPA and the same system considered in the untimed process calculus SOCK and in PEPA in [34].

10 Conclusion and Future Perspectives

In this paper we have addressed an inherent difficulty of modelling studies, namely that we lack certainty about details of the system which we are modelling. Very often this problem can be due to a lack of knowledge about rate parameters but it is also possible that we lack certainty about the function of some components of the system. This can be because we are undertaking a prospective modelling study of a still-to-be-constructed system. At this point decisions about the specification have not yet been finalised (and a quantitative modelling study can guide us in making the right decisions).

There are principally two ways to address these kinds of problems within a model. The first is to try to abstract away from the details of the components about which we are uncertain but such an approach risks missing too much detail and then our models will be unconvincing and their results will be inaccurate. The second is to try to include all of the detail of each possible component, presenting these as internal choices made by the model. Unfortunately, if we have included too much detail then we encounter the well-known state-space explosion problem where we simply do not have enough memory available to represent the detailed model or not enough time to compute the desired results. The model might be useful, if it helped us to think clearly, but we will always feel uncertain about it because we will be unable to test it or make predictions based on it.

The alternative which we have pursued here is to accept that we have *more than one model* to consider and that we need to apply our analysis to a related family of models. There are two attendant difficulties here. The first is that if we are to consider a large family of models then we surely want to generate these models automatically via a repeatable transformation. The `smc` compiler performs this function for the SRMC language, generating many PEPA models for a single input SRMC model. The second difficulty which we encounter here is that we have a daunting number of model evaluations to perform. For this approach to be feasible we need an evaluation mechanism which has low unitary cost. Fortunately the cost of solving initial value problems for systems of differential equations can extremely low.

We have applied these methods here to modelling Web Services and it seems that they suit this domain well because of its inherent uncertainty about binding sites and the attendant level of performance which we can expect to receive. However, we hope that our methods will also be useful beyond the domain of Web Services and that we may find many possible applications of the SRMC language in the future.

Acknowledgements. The authors are supported by the EU FET-IST Global Computing 2 project SENSORIA ("Software Engineering for Service-Oriented Overlay Computers" (IST-3-016004-IP-09)).

References

1. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
2. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, September 2005, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
3. Clark, A., Gilmore, S., Tribastone, M.: Service-level agreements for service-oriented computing. In: Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008), Pisa, Italy (June 2008) (to appear)

4. Clark, A., Gilmore, S., Tribastone, M.: Scalable analysis of scalable systems. In: Proceedings of Fundamental Approaches to Software Engineering (FASE 2009), New York, England (March 2009) (to appear)
5. OASIS Web Services Business Process Execution Language (WSBPEL) Technical Commitee. Web Services Business Process Execution Language Version 2.0 (April 2007)
6. Gilmore, S., Hillston, J., Ribaudo, M.: An efficient algorithm for aggregating PEPA models. IEEE Transactions on Software Engineering 27(5), 449–464 (2001)
7. Clark, A.: The ipclib PEPA Library. In: QEST [35], pp. 55–56
8. Tribastone, M.: The PEPA Plug-in Project. In: QEST [35], pp. 53–54
9. OASIS UDDI Specifications Technical Committee. Universal Description Discovery and Integration (UDDI), `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`
10. Clark, A., Gilmore, S.: State-aware performance analysis with eXtended Stochastic Probes. In: Thomas, N., Juiz, C. (eds.) EPEW 2008. LNCS, vol. 5261, pp. 125–140. Springer, Heidelberg (2008)
11. Clark, A., Duguid, A., Gilmore, S., Tribastone, M.: Partial evaluation of PEPA models for fluid-flow analysis. In: Thomas, N., Juiz, C. (eds.) EPEW 2008. LNCS, vol. 5261, pp. 2–16. Springer, Heidelberg (2008)
12. Bradley, J.T., Dingle, N.J., Gilmore, S.T., Knottenbelt, W.J.: Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. In: Kotsis, G. (ed.) Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, University of Central Florida, October 2003, pp. 344–351. IEEE Computer Society Press, Los Alamitos (2003)
13. Bradley, J., Dingle, N., Gilmore, S., Knottenbelt, W.: Extracting passage times from PEPA models with the HYDRA tool: A case study. In: Jarvis, S. (ed.) Proceedings of the Nineteenth annual UK Performance Engineering Workshop, July 2003, pp. 79–90. University of Warwick (2003)
14. Clark, A., Gilmore, S.: Evaluating quality of service for service level agreements. In: Brim, L., Leucker, M. (eds.) Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems, Bonn, Germany, August 2006, pp. 172–185 (2006)
15. Heimsund, B.-O.: MTJ: Matrix Toolkit for Java, `http://ressim.berlios.de/`
16. Bradley, J., Gilmore, S.: Stochastic simulation methods applied to a secure electronic voting model. Electr. Notes Theor. Comput. Sci. 151(3), 5–25 (2006)
17. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81(25), 2340–2361 (1977)
18. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. Journal of Physical Chemistry 104, 1876–1889 (2000)
19. CompBio Group, Institute for Systems Biology. ISBJava, `http://magnet.systemsbiology.net/software/ISBJava/`
20. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, September 2005, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
21. odeToJava library, `http://www.netlib.org/ode/odeToJava.tgz`
22. Dingle, N.J., Harrison, P.G., Knottenbelt, W.J.: HYDRA: HYpergraph-Based Distributed Response-Time Analyzer. In: Arabnia, H.R., Mun, Y. (eds.) PDPTA, pp. 215–219. CSREA Press (2003)

23. Hillston, J.: Tuning systems: From composition to performance. The Computer Journal 48(4), 385–400 (2005); The Needham Lecture paper
24. Hillston, J.: Process algebras for quantitative analysis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), Chicago, June 2005, pp. 239–248. IEEE Computer Society Press, Los Alamitos (2005)
25. Gilmore, S., Hillston, J., Ribaudo, M., Kloul, L.: PEPA nets: A structured performance modelling formalism. Performance Evaluation 54(2), 79–104 (2003)
26. Hillston, J., Ribaudo, M.: Modelling mobility with PEPA nets. In: Aykanat, C., Dayar, T., Körpeoğlu, İ. (eds.) ISCIS 2004. LNCS, vol. 3280, pp. 513–522. Springer, Heidelberg (2004)
27. Gilmore, S., Hillston, J., Kloul, L., Ribaudo, M.: Software performance modelling using PEPA nets. In: Proceedings of the Fourth International Workshop on Software and Performance, Redwood Shores, California, USA, January 2004, pp. 13–24. ACM Press, New York (2004)
28. Brodo, L., Degano, P., Gilmore, S., Hillston, J., Priami, C.: Performance evaluation for global computation. In: Priami, C. (ed.) GC 2003. LNCS, vol. 2874, pp. 229–253. Springer, Heidelberg (2003)
29. Calder, M., Gilmore, S., Hillston, J.: Automatically deriving ODEs from process algebra models of signalling pathways. In: Plotkin, G. (ed.) Proceedings of Computational Methods in Systems Biology (CMSB 2005), Edinburgh, Scotland, April 2005, pp. 204–215 (2005)
30. Geisweiller, N., Hillston, J., Stenico, M.: Relating continuous and discrete PEPA models of signalling pathways. Theor. Comput. Sci. 404(1-2), 97–111 (2008)
31. Zhao, Y., Thomas, N.: Approximate solution of a PEPA model of a key distribution centre. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 44–57. Springer, Heidelberg (2008)
32. Duguid, A.: Coping with the parallelism of BitTorrent: Conversion of PEPA to ODEs in dealing with state space explosion. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 156–170. Springer, Heidelberg (2006)
33. Gilmore, S., Tribastone, M.: Evaluating the scalability of a web service-based distributed e-learning and course management system. In: Bravetti, M., Núñez, M.T., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 214–226. Springer, Heidelberg (2006)
34. Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 204–221. Springer, Heidelberg (2008)
35. Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007), Edinburgh, Scotland, UK, September 2007. IEEE Computer Society, Los Alamitos (2007)

A Detailed Results

In this section we depict some of the graphs that we have produced from our example models which have not been shown in the main text. We include these here for completeness because it is sometimes the case that one can see the significance of one graph only in comparison to others.

Figure 19 shows some more of the candle stick graph possibilities which were not shown in Section 6. Figures 20 and 21 and shows all of the *time graphs* which plot probability of completion within the given time against experiment number.

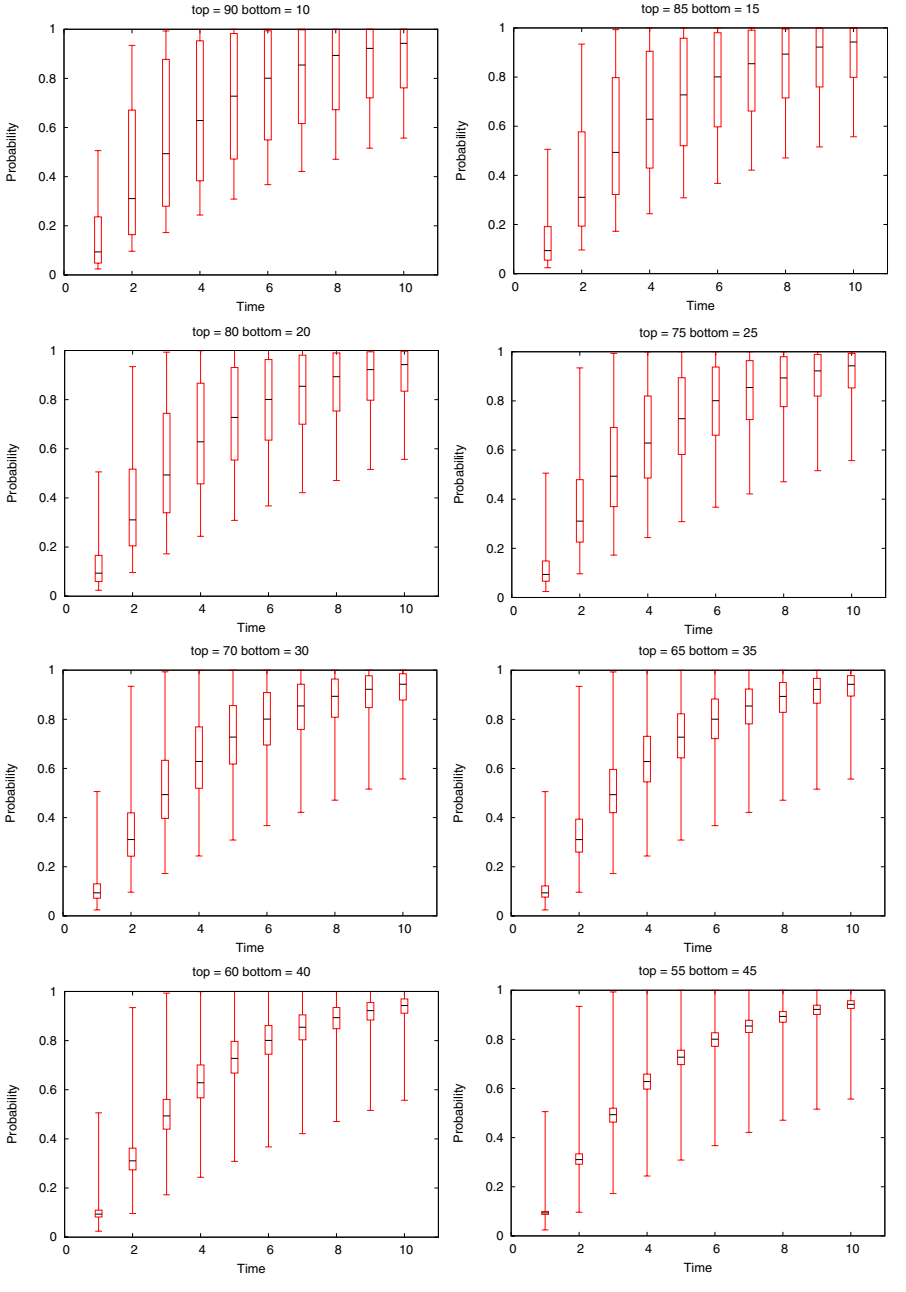


Fig. 19. Candlestick graphs showing the probability of completion ranging over all of the experiments performed automatically from the SRMC model

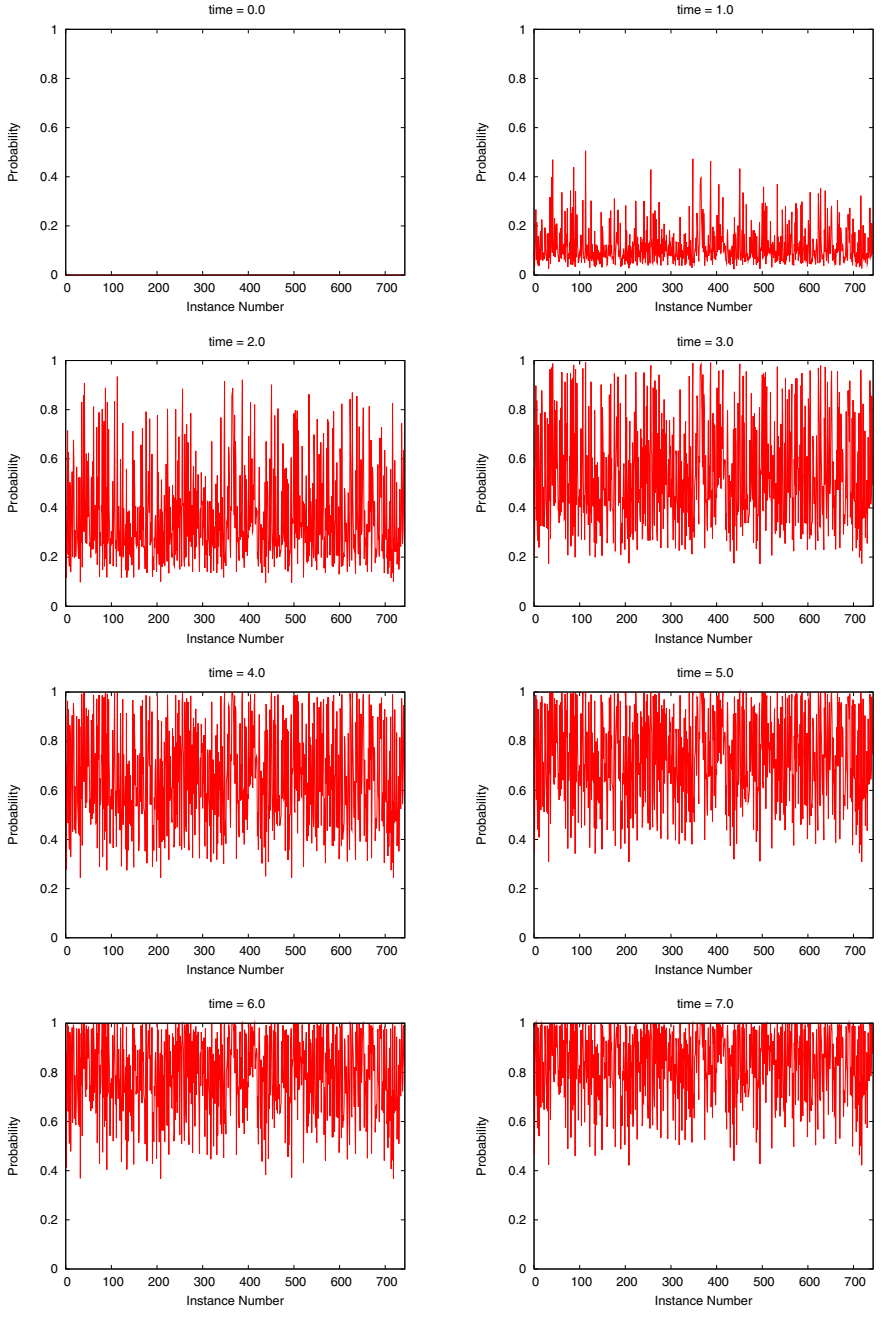


Fig. 20. Time graphs showing the probability of completion within the given times ranging over all of the experiments performed automatically from the SRMC model

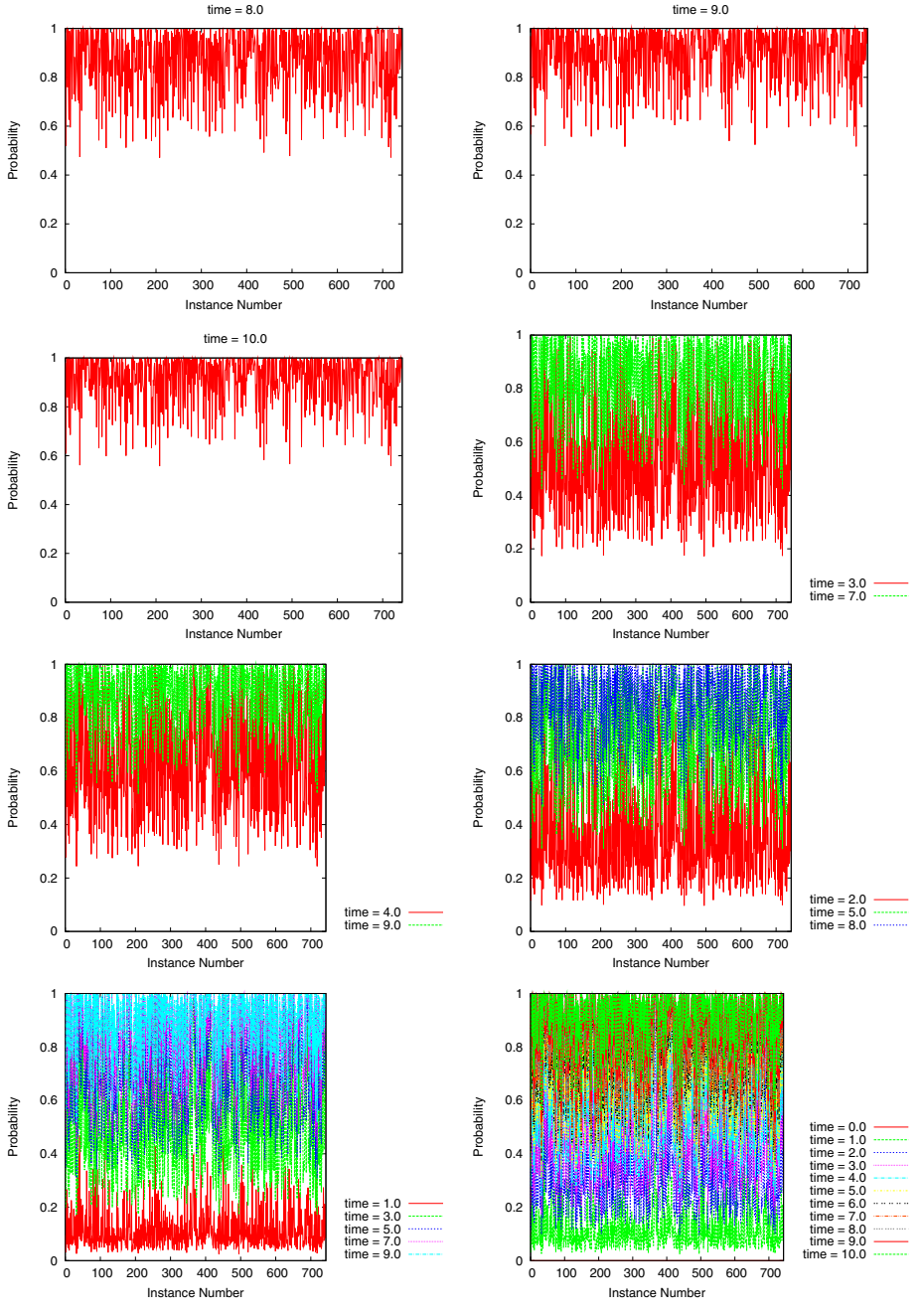


Fig. 21. Time graphs showing the probability of completion within the given times ranging over all of the experiments performed automatically from the SRMC model

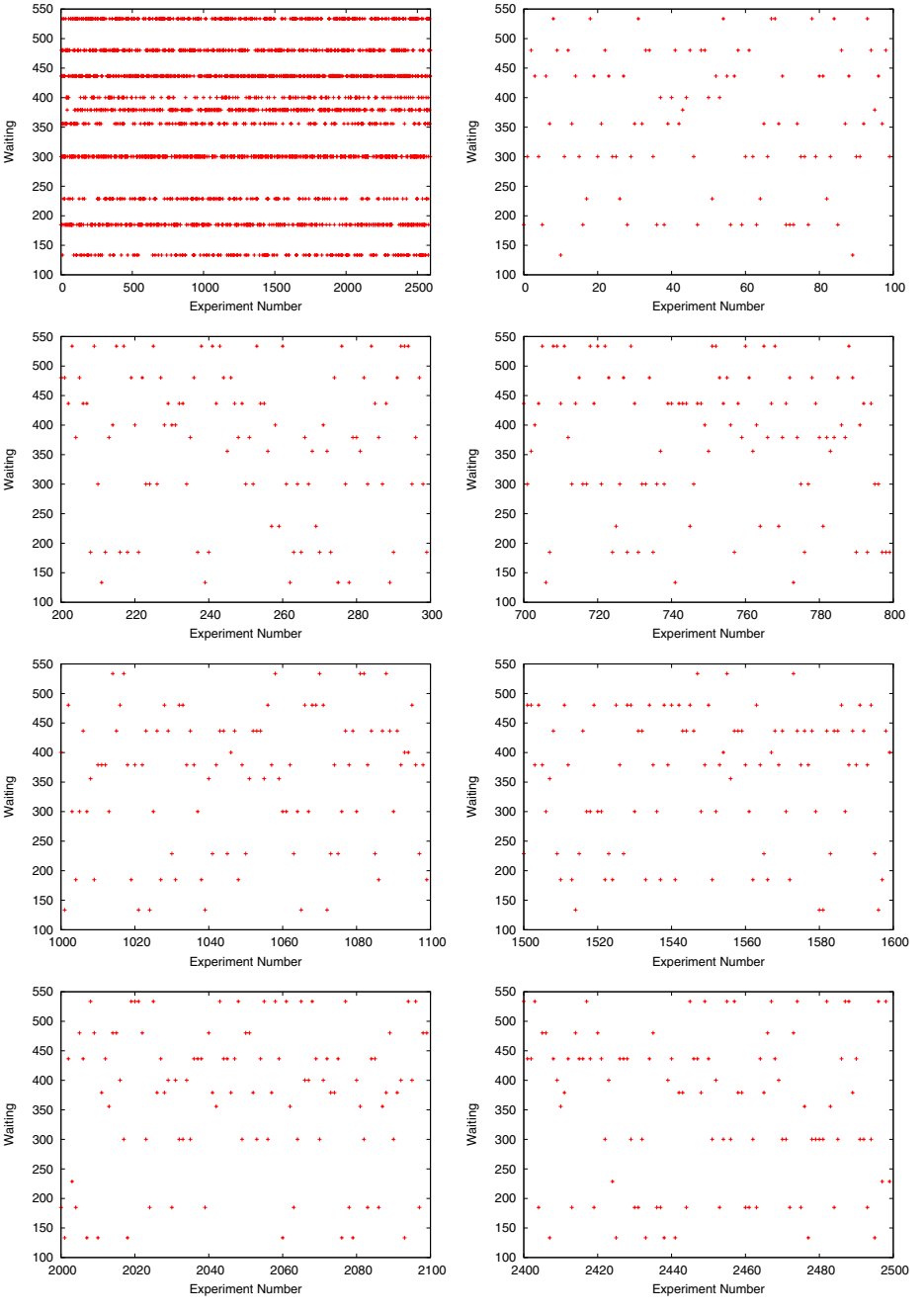


Fig. 22. Selected experiment number interval graphs for the SRMC model ODEs