Scalable Analysis of Scalable Systems

Allan Clark, Stephen Gilmore, and Mirco Tribastone

The University of Edinburgh, Scotland

Abstract. We present a systematic method of analysing the scalability of large-scale systems. We construct a high-level model using the SRMC process calculus and generate variants of this using model transformation. The models are compiled into systems of ordinary differential equations and numerically integrated to predict non-functional properties such as responsiveness and scalability.

1 Introduction

Very often, our ability to build complex software systems outstrips our ability to plan and carry out rigorous analysis which predicts the behaviour of these systems under conditions of increasing load. This situation is unsatisfactory because it leads to systems being deployed in active use with no real assurance of graceful degradation of service as the user population grows. Because we cannot rely on them to provide service in times of greatest need, such systems are as unreliable in practice as ones which contain programming errors.

The crippling blow which strikes when trying to scale discrete-state models to represent user populations of significant size is the well-known problem of state-space explosion. The discrete-state representation demands memory in quantities which grow too quickly for us to be able to meet these demands for long. A bold alternative is to abandon discrete-state representations and take our models to the continuous-space world using fluid-flow analysis [1]. This allows us to represent and analyse large-scale systems with modest requirements on memory and time.

Modelling large populations of users is seldom the only difficulty which we encounter with large-scale systems. Scalable systems need to be resilient to changes in the underlying operational conditions. For this reason they are often structured with critical services replicated on several hosts in order for the system to continue to function when some of these hosts fail. It is very unusual indeed for all of the hosts to have identical performance profiles. It is instead quite common for them to be running different versions of the software services. Some will be running an older version, others the latest. Some sites will have disabled certain features, others not.

As if the above did not already give us enough challenges we also need to address the issue that large-scale systems are dynamic. Hosts providing one service may be taken down and redeployed to provide a different service. Some hosts will fail and might not be replaced if they were thought to be underused. New hosts will be sourced, purchased and brought online where the need is

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 1–17, 2009.

perceived to be greatest. We would like our modelling study and our analysis results to be robust in the face of possible changes such as these.

In this paper we work with a process calculus which can be used for modelling problems such as these. The Sensoria Reference Markovian Calculus (SRMC), as described in [2], is a high-level modelling language which can be used for quantitative analysis of systems from the small scale to the large scale. Small-scale models in SRMC are mapped to continuous-time Markov chains and large-scale models are mapped to systems of differential equations. The SRMC language supports structured modelling via namespaces which accompany a novel mechanism for specifying uncertainty about binding. This is used to represent the inherent uncertainty about evaluation sites which is found generally in distributed computing and specifically in service-oriented computing where services are replicated across several hosts to provide scalability and robustness. Model transformations are used to capture changes in service administration leading to new hosts being commissioned or old ones being decommissioned.

The SRMC language is supported by a framework for experimentation and analysis which allows SRMC modellers to define their model together with a set of transformations. Software tools for SRMC generate the instances of the model which can be obtained through making different binding decisions. Once these binding decisions have been made the models can be expressed in a

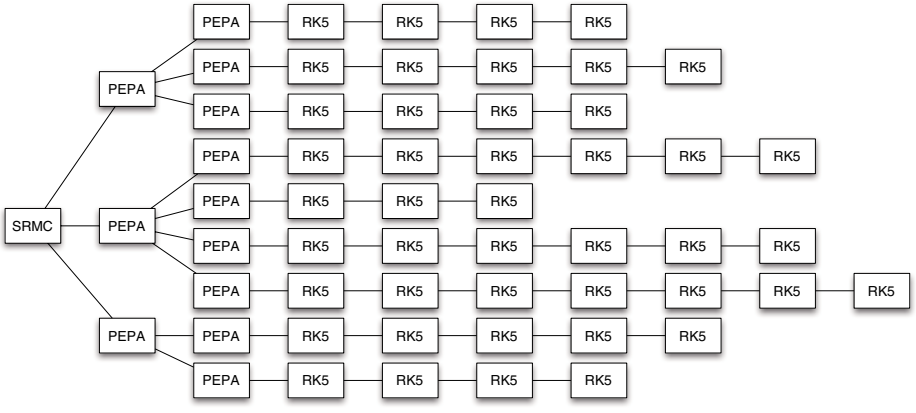

Fig. 1. The evaluation model. The single SRMC model in the illustration above gives rise to three PEPA models when possible bindings to services are considered. These three PEPA models become nine when model transformations are applied. Each of these nine models is evaluated between three and seven times in order to consider all of the possible assignments of rate values to rate parameters. This leads to forty-four systems of coupled ordinary differential equations in all and the same number of runs of a fifth-order Runge-Kutta (RK5) numerical integrator. These solve the initial value problem for each system of ODEs and give a time-series plot of the number of components of each type in the SRMC model as a function of time up to a finite time horizon. The results from these forty-four runs are combined to form the analysis results for the SRMC model.

simpler modelling language, Performance Evaluation Process Algebra (PEPA) [3,4]. Variants of each of these PEPA models are obtained by applying the transformations supplied. All models thus generated are evaluated for a range of numerical parameter values and the results from the individual runs are combined to deliver an evaluation of the model as a whole. Figure 1 illustrates the evaluation model for SRMC.

The novel contribution of the present paper is the use of model transformation to automatically generate a family of related models from a single SRMC source. In addition, this paper presents the first application of the fluid-flow analysis invented for PEPA to large-scale SRMC models. This latter innovation caused us to develop a supporting software infrastructure for aggregating the analysis results from the family of related models which we generate.

Structure of this paper: Section 2 presents our modelling concepts and introduces the ideas behind the process calculus which we use. In Section 3 we describe the features of the SRMC calculus and relate it to a simpler calculus without explicit support for dynamic binding, PEPA. In Section 4 we present the language of model transformations which we use. Our case study of a "virtual university" is presented in Section 5. The results are in Section 6. Software tool support is crucial for generating models and for managing the experimentation process; our software tools are described in Section 7. Section 8 describes related work and Section 9 presents conclusions.

2 Modelling Concepts

We are concerned here with non-functional properties of systems, specifically quantitative aspects such as performance. We investigate these properties using high-level models built from model components.

Model components are of two kinds, *behavioural* and *virtual*. A behavioural component cycles through a lifetime of timed activities offering sometimes only a single possible next activity and sometimes a choice between several alternatives. A virtual component does not perform activities but simply introduces a name into the model which we can use when querying the model later.

The kind of analysis which we will perform on the SRMC models in this paper tells us the expected number of behavioural components of every kind at all points in time. The expectation of the virtual components can be calculated from the expectations of the behavioural components by evaluating the defining expressions of the virtual components.

Because our interest is in quantitative modelling our models will contain rates and probabilities. Because we deal with large-scale systems with replicated components we also have *arrays* of behavioural components. For example, if C is a component then $C[15]$ is an array of fifteen independent copies of this component. We use these arrays to represent capacity (e.g. a pool of servers) or workload (e.g. a population of clients).

Our models will contain two kinds of numerical parameters, *certain* and *uncertain*. Certain parameters are bound to a single value: uncertain parameters

are bound to a set of possibilities. An uncertain parameter which is a probability might be bound to $\{0.4, 0.5, 0.6\}$, meaning that we should consider each of the elements of this set as a possible value for the parameter. Rates may be uncertain also, as may array sizes. Binding sites may be uncertain, and we choose one from a set of behavioural components.

We use model transformation to generate a family of related models from a single SRMC model. We think of these as plausible modifications of the original system which represent what the system might become after foreseeable reconfiguration or maintenance. Each of these generated models is "close" to the original model in the sense that they can be obtained by the application of a single transformation drawn from a set of possible transformations.

3 The Calculus

We work with the Sensoria Reference Markovian Calculus (SRMC), as described in [2]. SRMC allows modellers to structure their models using *namespaces*, separating components which have similar structure. To illustrate the SRMC language we will give an example of a simple system which consists of a process accessing one of two disks, A or B.

We first describe disk A which has occasional failures and has just two states, *failed* and *working*. When failed the disk must be repaired before more data can be read or written. Reads and writes are thought to be nearly equally likely: the probability that the I/O operation is a read is p_r. Failures occur somewhere between once every thousand disk operations ($p_f = 0.001$) and once every two hundred ($p_f = 0.005$). We will consider three sample values in this range. Rates λ and μ dictate the rates at which repairs and reads and writes take place. Disk A would be described in the SRMC syntax as shown below.

```
DiskA::{
  lambda = 0.3; mu = 1400; // rates
  p_r = { 0.4, 0.5, 0.6 }; // probability of a read
  p_f = { 0.001, 0.003, 0.005 }; // probability of failure

  Failed = (repair, lambda).Working;
  Working = (read, (1 - p_f) * p_r * mu).Working
        + (write, (1 - p_f) * (1 - p_r) * mu).Working
        + (fail, p_f * mu).Failed;
  Unavailable = [ Failed ];
};
```

This namespace has two behavioural components, `Failed` and `Working` and one virtual component `Unavailable`. The virtual component introduces the concept of "unavailability" to our model. In this case a disk is unavailable only if it has failed.

We can think of the above as a high-level schema representing nine concrete models differing only in the values assigned to the probabilities p_r and p_f. In the

first of the concrete models p_r has the value 0.4, and p_f is 0.001. In the ninth p_r is 0.6, and p_f is 0.005. In between all of the other possible assignments of values to p_r and p_f have been enumerated.

Disk B is slower than disk A. Failures occur more frequently and they take longer to repair. In addition disk B has a sleep mode which it enters to save power. The SRMC description is below.

```
DiskB::{
  lambda = 0.1; mu = 1200; // lower rates for the slower device
  p_r = { 0.4, 0.5, 0.6 }; // same probability of a read
  p_f = { 0.01, 0.03, 0.05 }; // higher probability of failure
  gamma = 0.001; delta = 0.001; // rates for sleep and wake

  Failed = (repair, lambda).Working;
  Working = (read, (1 - p_f) * p_r * mu).Working
        + (write, (1 - p_f) * (1 - p_r) * mu).Working
        + (fail, p_f * mu).Failed
        + (sleep, gamma).Offline;
  Offline = (wake, delta).Working;
  Unavailable = [ Failed + Offline ];
};
```

This too is a high-level schema representing nine concrete models. However, it introduces a different notion of unavailability. Here a disk is unavailable if it has failed or is offline. The virtual component `Unavailable` is defined to be the arithmetic sum of the number of disks which have failed plus the number of disks which are offline. Notice that the symbol "`+`" is overloaded in SRMC. In a behavioural component "`+`" denotes choice. In a virtual component "`+`" denotes arithmetic sum. Here we will add the expected number of failed disks and the expected number of offline disks to get the expected number of unavailable disks. Virtual components are syntactically distinguished because they consist of a defining expression enclosed in square brackets.

The disk which is in use in the system is either disk A or disk B. To describe this in SRMC we introduce another namespace, `Disk` which can stand for either A or B.

```
Disk::={ DiskA, DiskB };
```

The top-level composition of components in our example here introduces a computational process which reads and writes. The disk which is used is initially in its working state.

```
System = Process::Idle <read, write> Disk::Working;
```

In all this SRMC model represents eighteen simpler concrete models. In nine of these disk A is being used. In the other nine disk B is being used. In evaluating this SRMC model we separate out two groups of models. In the first of these disk A is being used, and disk B is not represented at all—the top-level composition evaluates to `Process::Idle <read, write> DiskA::Working` and

the entire `DiskB` namespace is discarded. In the second group of models disk B is being used and disk A is not represented at all—the top-level composition becomes `Process::Idle <read, write> DiskB::Working` and the entire `DiskA` namespace is discarded.

We then perform a *parameter sweep* across the possible assignments of values to model parameters in each group. This will require us to evaluate the version of the model with the main disk nine times, and the version of the model with the spare disk nine times also. Finally, we combine the results.

It is important to understand that the model configuration is fixed during model evaluation. That is, we will investigate the behaviour of the model up to a finite time horizon and during this time interval the model configuration will not change. This ability to divide the initial SRMC model up into a collection of simpler static models is an important factor in making our analysis scale to large models.

The simpler models which are generated in the parameter sweep which is performed after resolution of binding do not have namespaces and do not have uncertain parameters. The consequence of this is that they can all be expressed in Performance Evaluation Process Algebra (PEPA) [3,4]. PEPA has both a discrete-state stochastic Markovian semantics [5] and a continuous-state sure differential equation semantics [1]. We have considered the evaluation of SRMC models using the Markovian semantics for PEPA in an earlier paper [2] and we use the differential equation semantics here because our concern is with evaluating large-scale systems with many users and many replicated services.

Once all of the separate instances of the generated PEPA models have been analysed we collate the results into what is now a database of results. This database can be used to select and display various results from results relating to a single configuration or subset of all configurations to results pertaining to the entire results space. The latter allows such queries as: "What is the worst case scenario of long term expected number of unavailable disks" and "Give a listing of all configurations which fail to satisfy a given throughput of read operations".

4 Model Transformation

The previous section relates the process of numerically evaluating an SRMC model. This included generating PEPA models after resolving dynamic binding. However, we also wish to investigate related models, reachable by an application of a model transformation, in order to incorporate possible changes which may occur. The grammar in Figure 2 defines transformation rules. The description presented here is sufficiently general that it may be applied at either the SRMC or the PEPA level.

A rule is specified by providing a *pattern* which should match some subcomponent of the model and a corresponding replacement, which is syntactically also a pattern. A transformation rule may contain *pattern variables* denoted by a question mark followed by a name. A pattern variable will match anything

rule	:= *pattern*`==>`*pattern*	rules
pattern	:= `?`*name*	variable
	| *name*	named
	| *pattern* `<`*activities*`>` *pattern*	cooperation
	| *pattern* `[`*size*`]⟨[`*activities*`]⟩`	array
activities	:= ⟨`?`*name*`,`⟩*name**	concrete activities
size	:= `?`*name*	variable
	| *integer*	constant
	| *size binop size*	binary op
binop	:= $+ \mid - \mid \times \mid \div$	operators

Fig. 2. The grammar for transformation rules. The names which appear in the patterns are component identifiers. The names which appear in activity sets are activity names. The names which are used in size expressions denote the integer values which are used in dimensioning arrays of components.

which may appear in the given position, so when it occurs in the place of a component then it will match any component. If the replacement refers to a pattern variable then whatever was matched against is inserted at that place in the replacement.

A list of activities may contain a pattern variable together with several other concrete activity names. If this is the case the pattern variable is set to those names which are not given concretely, however we only match the given set of activities if the concrete activities are contained within the set.

The transformation `P <a> Q ==> P <a,b> Q` adds activity `b` to the cooperation set. This rule uses no pattern variables and so will only match against the cooperation `P <a> Q`. Generally pattern variables are used as `?Q` here in the rule `P <a> ?Q ==> P <a,b> ?Q`. This will match the cooperation of `P` with any component including one which is itself a cooperation or a component array. Here we match against two cooperating arrays of `P` and `Q` components, remove one `P` and add a `Q` instead: `P[?m] <a> Q[?n] ==> P[?m - 1] <a> Q[?n + 1]`.

This style of pattern matching is used with *redeployable components* where we wish to analyse our system with different numbers of components deployed in each role. For example, file servers may be redeployed as web servers. However the pattern more commonly abstracts over the cooperation set as in the pattern: `P[?m] <?a> Q[?n] ==> P[?m - 1] <?a> Q[?n + 1]`.

Finally a common pattern is to remove some activities from a cooperation set. The following pattern matches any component cooperating with a `P` component over the activity `b` and removes it from the cooperation set allowing the `P` component (and the other cooperating component) to perform the activity `b` independently. Note though that any other activities in the cooperation set are preserved. This is written as `P <?a, b> ?Q ==> P <?a> ?Q`.

5 Case Study

To illustrate the above ideas we consider as an example a distributed e-learning and course management system. The system is to allow students to enrol in

courses even when studying remotely. One of the quantitative issues of concern here is whether or not the system will scale well enough to cope with increased demand from a larger population of student users.

5.1 The Servers

In this example we consider a fictional virtual university which has two university sites in the University of Edinburgh and Imperial College, London. Each site has an HTTP server where students can download course materials, multimedia content, and other courseware. Each has an FTP server where students can upload project materials and coursework for assessment. The HTTP and FTP servers may fail independently and, because each is running other services as well, availability of the servers varies.

```
Edinburgh::{
   mu = 0.0001; gamma = 0.125; // rates of fail and repair
   avail = {0.6,0.7,0.8,0.9,1.0}; // availability of the server
   phi = 10.0; psi = 7.0; // rates for download and upload

   // The HTTP server
   Http::{
      Idle = (download, avail * phi).Idle
           + (fail, mu).Broken;
      Broken = (repair, gamma).Idle;
   };

   // The FTP server
   Ftp::{
      Idle = (upload, avail * psi).Idle
           + (fail, mu).Broken;
      Broken = (repair, gamma).Idle;
   };
};
```

The servers at Imperial are similar in functionality to those in Edinburgh however they differ in their performance characteristics, specifically with respect to the rates at which failures occur and the rates at which downloads and uploads occur.

```
Imperial::{
   mu = 0.006; gamma = 0.125; // failures are more likely
   avail = {0.6,0.7,0.8,0.9,1.0}; // availability is the same
   phi = 20.0; psi = 15.0; // download and upload are faster

   // The HTTP server
   Http::{
      Idle = (download, avail * phi).Idle
```

```
            + (fail, mu).Broken;
       Broken = (repair, gamma).Idle;
   };

   // The FTP server
   Ftp::{
       Idle = (upload, avail * psi).Idle
            + (fail, mu).Broken;
       Broken = (repair, gamma).Idle;
   };
};
```

Of course further servers may be added, in our results given in Section 6 we added a further server which fairly services both HTTP and FTP requests at the same rate.

5.2 The Clients

We characterise different types of user of the system. The first, Harry, connects relatively frequently, and uploads or downloads once each session.

```
Harry::{
  connect_rate = { 0.01, 0.02, 0.03 };
  disconnect_rate = 1.0;
  download_rate = { 0.01, 0.02, 0.03 };
  upload_rate = 1.0;

  Idle = (connect, connect_rate / 2).Upload
           + (connect, connect_rate / 2).Download;
  Upload = (upload, upload_rate).Disconnect;
  Download = (download, download_rate).Disconnect;
  Disconnect = (disconnect, disconnect_rate).Idle;
  Uploading = [ Upload ];
  Downloading = [ Download ];
  Inservice = [ Uploading + Downloading ];

};
```

The second type of user, Sally, connects relatively infrequently and downloads more than uploading.

```
Sally::{
  connect_rate = { 0.009, 0.0095, 0.01 };
  disconnect_rate = 0.5;
  download_rate = { 0.01, 0.02, 0.03 };
  upload_rate = 0.2;
```

```
  Idle = (connect, connect_rate / 3).Upload
           + (connect, connect_rate / 3).Download1
           + (connect, connect_rate / 3).Download2 ;
  Upload = (upload, upload_rate).Disconnect;
  Download1 = (download, download_rate).Download2;
  Download2 = (download, download_rate).Disconnect;
  Disconnect = (disconnect, disconnect_rate).Idle;

  Uploading = [ Upload ];
  Downloading = [ Download1 + Download2 ];
  Inservice = [ Uploading + Downloading ];
};
```

As with the servers further clients may be added as required, in our results we added a further client who was likely to perform either two uploads or two downloads with each connection.

5.3 The Model Configuration

The clients are either like Harry or like Sally.

```
Client::= { Harry, Sally };
```

The HTTP server which is used is either the Edinburgh server or the Imperial server, and analogously for the FTP servers.

```
Http::= {Edinburgh::Http, Imperial::Http};
Ftp ::= {Edinburgh::Ftp, Imperial::Ftp };
```

There are between three and six servers at each site. Each server has an allocation of twenty threads to offer. There is a very large pool of clients.

```
servers = {3, 4, 5, 6};
threads = 20;
clients = 100000;
```

The entire model consists of an array of clients uploading and downloading from an array of servers, with multiple threads on each.

```
Client::Idle[clients] <download, upload>
    ( Http::Idle[servers * threads] ||
      Ftp::Idle[servers * threads] )
```

5.4 Transformations

The transformations used in this model relate to the redeployment of a server. This means that a server currently being used as an HTTP server can be redeployed as an FTP server or vice-versa. We use this to test a particular service configuration's ability to adapt to varying client behaviours. Recall that one configuration is a set of name space choices. In our given example this means that

one particular configuration is a selection of a university to supply the HTTP server, a university to supply the FTP server and finally a client representing average client behaviour. Here we call a service configuration the part of the configuration which specifies the two servers in use.

One such configuration is *Edinburgh*, *Imperial* and *Harry*. Suppose we measure this and we find that the average number of waiting clients is acceptably low, but in the configuration *Edinburgh*, *Imperial* and *Sally* where we have changed the client behaviour the system behaves poorly. We may see that it behaves poorly because the uploading clients are not serviced fast enough. In practice if such a situation arose one response would be to redeploy one of the HTTP servers as an FTP server. For each configuration, which corresponds to a single PEPA model, we use transformations to obtain three PEPA models; the base configuration model, the model with one HTTP server redeployed as an FTP server and the model with one FTP server deployed as an HTTP server. The first transformation would tell us how the system behaves in the configuration *Edinburgh*, *Imperial* and *Sally*, with one HTTP server redeployed as an FTP server. Without this transformation we noted that the performance was poor because uploading clients were not serviced often enough, with this transformation though it may be that the system performs satisfactorily. In this case we would know that the system configuration is robust with respect to changing client behaviour and thus the system may be recommended. The transformation rule used to obtain this is:

```
Http::Idle[?m * threads] || Ftp::Idle[?n * threads] ==>
  Http::Idle[(?m - 1)*threads] || Ftp::Idle[(?n + 1)*threads]
```

and similarly for redeploying in the reverse direction.

5.5 Lazy Results

In the example case study we have used a strict semantics for results generation, that is; all the results were computed before any were viewed by the user. However an alternative semantics allows the user to generate only those results which are inspected. In the previous example scenario in which the configuration: *Edinburgh*, *Imperial* and *Sally* under-performed because the uploading clients were not serviced enough we would be unlikely to inspect the result obtained by applying the transformation which redeploys an FTP server as an HTTP server since this would only exacerbate the situation. In this case we could avoid computing all the results associated with that particular model – there are more than one set of results for each model because the model is solved several times corresponding to the varying rates used within that model instance. Lazy results can help in the development of a model since often only a small set of the results are viewed between each update of the entire model. However lazy results are limited to local obervations, that is results which only depend on a subset of the entire results space. Many results such as overall average response-time or worst-case scenario depend on all of the results and in these case lazy results will behave in the same way as strict results.

6 Results

We analysed our model on a typical desktop computer. In our first run we analysed over 1000 configurations within fifteen minutes. In our second run we increased the variability of the rates involved and performed over 4500 in six hours. Figure 3 provides a selection of the graphs which were generated from the second run.

The main result metric we used was the measure of the expected number of clients currently in the proceess of uploading, downloading or either of the two. We term this *in service*. This gives us a measure of how many are generally competing for the resources of the servers and – in combination with the throughput of download/upload events – how long each client can expect to wait to be served.

Graph (a) plots for a single configuration the number of clients in service against time. Each line corresponds to a single permutation of the variable rates which are used with the particular configuration (*Edinburgh*, *Edinburgh* and *Harry*), without any redployment of servers. We see that in the long run the expected number of clients waiting varies from just over thirty thousand to over eighty thousand of a total of one hundred thousand users. This shows that there is substantial variation in system performance caused only by varying the rates within a single configuration.

In graphs, (b), (c) and (d) respectively we have plotted for a single configuration the effect that redeploying a server has on the number of clients either uploading, downloading or either of the two. These three graphs all refer to the first configuration (*Edinburgh*, *Edinburgh* and *Harry*). In the graphs (b) and (c) it is shown that redeployment gives a large benefit to the recipient of the redeployment. So in the case that an HTTP server is redeployed as an FTP server the number of currently uploading clients (in graph (b)) falls almost to zero. Whereas redeployment of an FTP server reduces the number of currently downloading clients (graph (c)). In this particular configuration we note that redeploying an FTP server has only a small benefit to the number of downloaders since this is low anyway (the red/middle line in graph (c)). However redeploying an HTTP server has a large benefit in reducing the number of uploaders (the blue/bottom line in graph (b)). Finally from graph (d) we see that either redeployment increases the total number of in service clients for this particular configuration. However this is not always the case as in graph (e) in which the configuration (*Edinburgh*, *Imperial* and *Sally*) is considered the number of overall in service clients reduces when an FTP server is redeployed.

The graphs (f), (g) and (h) plot the experiment number against the total number of clients in service, uploading and downloading respectively. In each graph the experiments which correspond to an initial configuration (without any redeployment of servers) are shown in red on the left. The experiments in which the configuration has been altered by redeploying an FTP server as an HTTP server are plotted in green (in the middle) and finally those in which an HTTP server has been redeployed as an FTP server are plotted on the right in blue. From graph (f) we see that the redeployment of a server has a relatively

minor effect on the total number of clients in service. Redeploying an FTP server does worsen both the worst case scenario and the best case scenarios and in general decreases the performance by our metric – though not for all individual configurations as we have already seen from graph (e). Redeploying an HTTP server actually gives better worse and best case scenarios but performance in the general case is mildly impaired. From graphs (g) and (h) we see that the redeployment of a server generally has the intuitive effect. That is if we redeploy an FTP server the number of clients downloading decreases while the number of clients uploading increases and vice-versa. However it is encouraging to note that there are some configurations in which the redeployment of an FTP server still results in very low numbers of uploaders and conversely the redeployment of an HTTP server still results in very low numbers of downloaders.

7 Software Support

The software support which is available for SRMC has its basis in the parameter sweep developed for PEPA [6] and implemented in IPC [7]. We use the Pepato library of the PEPA Eclipse Plug-in Project [8] to compile our generated PEPA models to systems of ODEs. The Pepato library gives us access to differential equation integrators [9,10]. The model transformation engine used for our process algebra models was developed for the present work. The software used here is available for download from `http://groups.inf.ed.ac.uk/srmc`.

8 Related Work

We have considered the distributed e-learning and course management system example previously. In [2] we considered the problem of how service-level agreements can be evaluated for service-oriented systems at all. In [11,12] we considered the scalability of such a system in the absence of possible modifications as generated through model transformation.

Considering the method of evaluation rather than the example, the quantitative modelling approaches which seem similar to ours in spirit are PEPA itself, stochastic Petri nets, and PRISM model-checking. We consider each of these in turn, giving attention to the way in which parameters can be varied, the type of results which can be computed, and the query languages which are available to query models.

PEPA Eclipse Plug-in. The analysis paradigm for PEPA as supported by the PEPA Eclipse Plug-in [8] is that we have a single model with rates which can be varied using the experimenter of the Plug-in. Models can be evaluated to determine utilisation, throughput and many other criteria. Queries are expressed using state filters.

IPC. The paradigm for PEPA supported by IPC [7] is that we have a single model and rates defined in the model can be overridden at evaluation time. Models can be evaluated to determine transient or steady-state behaviour.

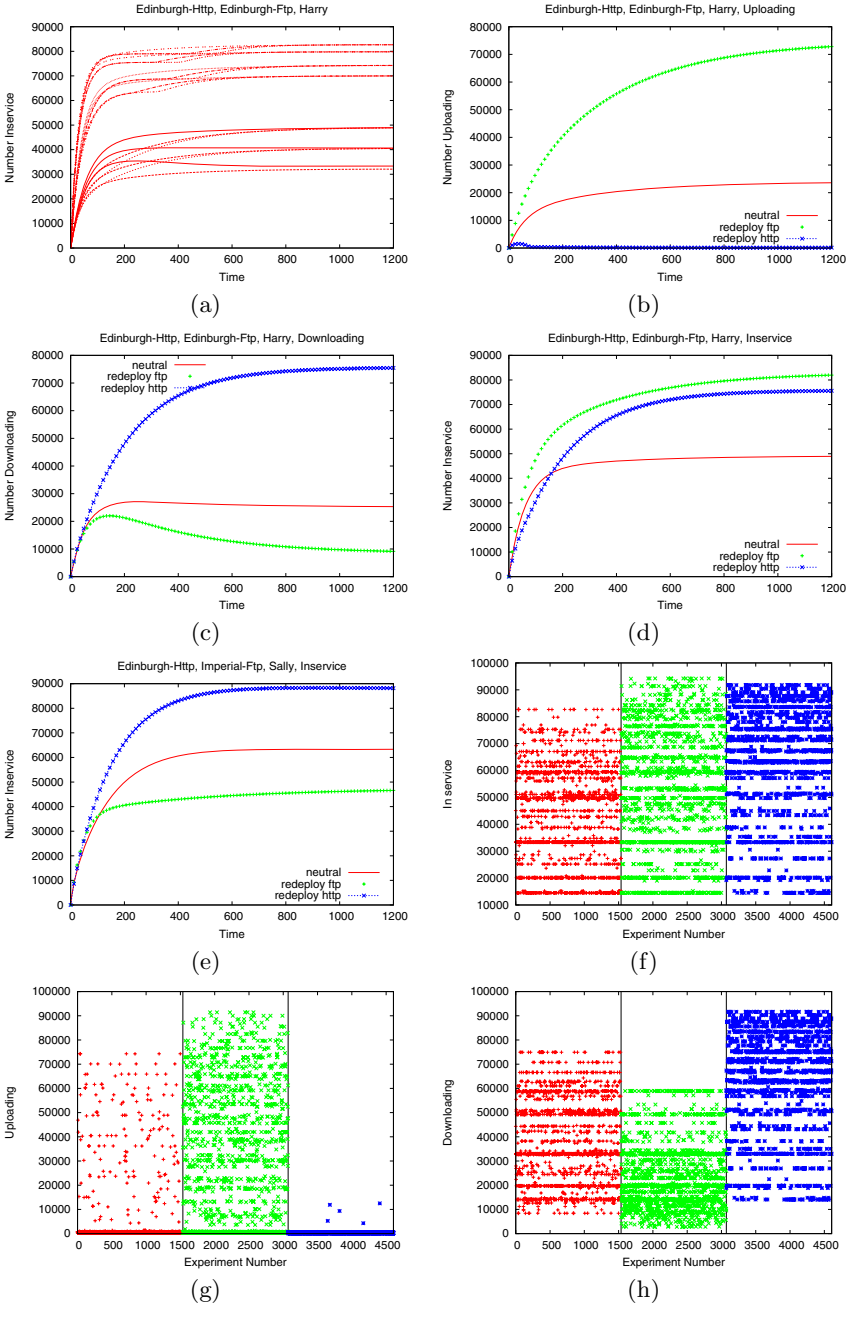

Fig. 3. Selection of generated graphs

Passage-time quantiles are computed. Queries are expressed in the language of eXtended Stochastic Probes (XSP) [13].

PIPE and PQE. Generalised Stochastic Petri net (GSPN) models are supported by the PIPE editor [14]. Here a single model with fixed rates is evaluated against many criteria. Queries are expressed as Performance Trees [15] built graphically in the Performance Query Editor (PQE) module for PIPE.

PRISM. The PRISM model-checker supports a language of reactive modules [16] together with a rich reward language. Queries are expressed in Continuous Stochastic Logic (CSL) [17]. Experimentation is supported by leaving constants in the model or the CSL formula undefined until the time of evaluation.

9 Conclusions

We have placed the emphasis here on modelling rather than models, and on evidence rather than fact. We start from a position of uncertainty about the configuration of the computational framework and consider a family of related models in an attempt to understand the sweep of possibilities. We believe that this position is a realistic one. In service-oriented computing the critical services are replicated across hosts so we have a choice of service instances, possibly modified by system administrators, performing at different rates. For these reasons, the SRMC calculus provides support for namespace selection, model transformation, and parameter sweep.

Model transformations work at the level of the PEPA model and can therefore be deployed as a means of analysing possible changes in one particular model. In this work we have used the ability to specify generic transformations in order to apply such a transformation to a large range of PEPA models generated from our SRMC model.

Collating results allows us to answer questions of a general nature about all configurations. In our case study there were few general statements that could be made because the transformations and rate variations provided substantial changes in performance. This is in itself a useful fact, that the system under consideration is sensitive to modifications in the running conditions. However we were able to ascertain worst and best case scenarios for the average number of currently downloading and currently uploading clients. Of particular note was that the number of waiting uploading clients can be all but eliminated through the redeployment of an HTTP server regardless of the initial configuration. We also saw that in general the redeployment of a server would most often benefit one kind of user (say uploaders) at the cost of denying some service capacity to the other (downloaders). This was also shown by the fact that the number of clients in service either as an uploader or a downloader was relatively less affected by redeployment of servers.

Model evaluation must be rapid to support the investigation of many alternative models. Fluid-flow analysis allows us to obtain meaningful results from a large family of models, at low computational cost. The result from a fluid-flow

analysis performed by numerically integrating a system of ordinary differential equations is precise and definitive. We need to evaluate our system of ODEs only once, not many times as would be needed for simulation models. This supports the scalability of our analysis: running many models once is feasible, running many models many times is less so.

Because fluid-flow analysis does not use a representation of the discrete state-space of the system we are not crippled by the state-space explosion problem, unlike any analysis which is based on continuous-time Markov chains. By building on the foundation provided by fluid-flow analysis and creating software tools which automate the generation of models by model transformation and the quantitative evaluation of these we hope to provide a strong basis for scalable analysis of scalable systems.

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.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
2. Clark, A., Gilmore, S., Tribastone, M.: Service-level agreements for service-oriented computing. In: Montanari, U., Corradini, A. (eds.) Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008), Pisa, Italy. LNCS. Springer, Heidelberg (2008)
3. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
4. Hillston, J.: Tuning systems: From composition to performance. The Computer Journal 48(4), 385–400 (2005); the Needham Lecture paper
5. Hillston, J.: Process algebras for quantitative analysis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 239–248. IEEE Computer Society Press, Chicago (2005)
6. 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, pp. 172–185 (August 2006)
7. Clark, A.: The ipclib PEPA Library. In: Harchol-Balter, M., Kwiatkowska, M., Telek, M. (eds.) Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 55–56. IEEE, Los Alamitos (2007)
8. Tribastone, M.: The PEPA Plug-in Project. In: Harchol-Balter, M., Kwiatkowska, M., Telek, M. (eds.) Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 53–54. IEEE, Los Alamitos (2007)
9. Ascher, U.M., Ruuth, S., Spiteri, R.: Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. Applied Numerical Mathematics 25(2-3), 151–167 (1997)
10. Dormand, J., Prince, P.: A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics 6(1), 19–26 (1980)

11. Gilmore, S., Tribastone, M.: Evaluating the scalability of a web service-based distributed e-learning and course management system. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 214–226. Springer, Heidelberg (2006)
12. Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 204–221. Springer, Heidelberg (2008)
13. 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)
14. Bonet, P., Lladó, C., Puijaner, R., Knottenbelt, W.J.: PIPE v2.5: A Petri net tool for performance modelling. In: 23rd Latin American Conf. on Informatics (CLEI 2007) (September 2007)
15. Suto, T., Bradley, J.T., Knottenbelt, W.J.: Performance Trees: A new approach to quantitative performance specification. In: MASCOTS 2006, 14th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 303–313 (August 2006)
16. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
17. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. 29(7), 1–18 (2003)