QoS Explorer: A Tool for Exploring QoS in Composed Services

Conrad Hughes
School of Informatics
University of Edinburgh
conrad.hughes@ed.ac.uk

Jamie Hillman
Department of Computing Science
Lancaster University
mail@jamiehillman.co.uk

Abstract

*This paper presents **QoS Explorer**, an interactive tool we have developed which predicts quality of service (**QoS**) of a workflow from the QoS characteristics of its constituents, even when the relationships involved are complex. This facilitates design and instantiation of workflows to satisfy QoS constraints, as it enables the user to discover and focus effort on the aspects of a workflow which most affect their primary QoS concerns, thus improving efficiency of workflow development. Further, the underlying model we use is more sophisticated than those of similar recent work [14, 2, 19], and includes processing of entire statistical distributions and probabilistic states (instead of the simple numeric constants used elsewhere) to model such* non-*constant variables as execution time.*

1 Introduction

Applications and business systems are now commonly provided as web services. Larger software systems are as a result increasingly constructed not monolithically, but as compositions of web services — *workflows*. "Workflow" here refers to a description of the composition of a set of web services in terms of the flow of control from service to service. This flow of control might be as simple as executing a series of services one after another, but many workflows are much more complex, involving parallel execution, conditionals and error handling; there are many specialist languages devoted to describing workflows as a result [20, 1, 8, 6, 17, 15, 4, 3].

A workflow might not specify a particular service instance to carry out each part of the task. Service function, type or interface could instead be specified, leading to greater flexibility: since in some cases there may be many services available that perform a required task, the most suitable service for each "slot" in the workflow may be selected later in order to optimise the system's non-functional characteristics (including QoS). This allows the workflow to be *instantiated* in different ways to satisfy a particular user's QoS requirements without altering its functional behaviour.

These QoS requirements may cover a wide range of non-functional characteristics including performance (e.g. time to complete), security (e.g. time between availability and application of security patches and antivirus updates), reliability (e.g. mean time to failure) and availability (e.g. mean uptime per year). These characteristics may be expressed in many different ways, and their names may have different meanings to different communities — so *ontologies* will prove benificial in specifications to express agreed definitions of QoS terms, including both circumstances and units of measurement. Several QoS specification languages and frameworks exist, including QML [9] and WSLA [16].

Given a specification of a particular service's QoS characteristics and a reference to an ontology that describes the included terms, the developer assembling a composite service can then select which of the available compatible services should be used.

Where a composition involves very few components and their relationship in the workflow is simple, choosing which component services to use should be relatively easy — for example, if performance is crucial in the application then the services with the best performance characteristics should be selected.

Where services for a non-trivial workflow are being selected in face of mutually conflicting requirements (e.g. low cost *and* high performance), this task becomes more difficult. Services running in nested structures of parallel and serial execution, with conditional branching and error handling can produce complex relationships. Complex relationships between the services in the workflow lead to a less intuitive relationship between the performance of the individual services and the performance of the composite service.

QoS Explorer can be used interactively to experiment with different characteristics of components in a workflow in order to see how these characteristics affect those of the whole. These experiments can then be used to help decide

which of a range of candidate services should be selected for the workflow, or to determine which components are the main cause of constraint violations. A range of potential uses for the tool is discussed later in the paper.

This introduction continues with more background on service composition and the idea of experimenting with compositions; Section 2 covers how we compute aggregate QoS for a workflow from information about its components; Section 3 shows QoS Explorer in action; and we conclude in Section 4 with an evaluation and placement of this work in context of other recent research in the area, as well as an outline of possible future work.

1.1 Service composition

One of the major themes of service-oriented architectures is that of building complex services out of simpler components, i.e. performing higher level tasks by creating "value-added" services out of pre-existing building blocks. This follows on from many earlier notions in software development, viz. components, code reuse, shared libraries, etc. Many new languages have been proposed to facilitate this kind of "programming in the large", including WS-BPEL [20] (the OASIS-standardised version of BPEL4WS [1], itself a merger of IBM's WSFL [8] and Microsoft's XLANG [6]), GSFL [17] (WSFL adapted to grid computing), WSCDL [15] (a W3C candidate recommendation), WSCL [4] and WSCI [3] (W3C notes preceding WSCDL). Approaches vary from declaratively specifying an executable sequence of operations through to more abstract conversation/choreography description. There is also a school which argues that all of these languages are redundant, and that we should simply be adapting existing scripting languages and RAD tools — Perl, PHP, Python, Ruby — to the task. For our purposes however, all that is really needed is a boxes-and-lines description of the workflow: many details, such as the nature of the data to be transferred, the type transformations involved in preparing one service's output for input to another, etc. are irrelevant to inferring aggregate behaviour of the whole. The kind of structures we do need to know about are the ordering constraints on the workflow — what can be run when — so the workflow description we use reflects that: our workflow elements include "run these operations in parallel and wait until they've all finished", "run these one after another", "run this if a certain condition is met", etc. This kind of high-level structural profile of a workflow could be automatically abstracted from many of the existing languages simply by stripping away the excess detail that they specify; Van Der Aalst et al. [21] give a good taxonomy of the fundamental structures involved.

In order to be able to compute the aggregate behaviour of these workflows, it is necessary that all services within the workflow agree on what they're measuring: if one is measuring cost in Yen while another is measuring it in Euro, or one is measuring "time to complete" from the client perspective (including network latency) while another is measuring it from the server perspective (no latency), it is clearly not valid to try aggregating these measurements — they're not the same quantity, or not the same units, etc. A common meaning must be agreed for all metrics being processed. QoSOnt [7] is our approach to providing a QoS ontology that provides the basis for agreement across components and projects.

When trying to compute the effects of composition on the measurable characteristics of a set of services — to infer, say, the total running time, total cost, maximum network bandwidth required, likelihood of failure — it is clear that each different workflow element (parallel-all, parallel-first ("start these services all at once but continue as soon as the first completes"), conditionals, series, etc.) will have a different effect on different characteristics. For example, when running a collection of services in series, the bandwidth requirement of the collection is the maximum of the services' individual bandwidth requirements, while the time to complete is the sum of all of their times to complete. In contrast, when running the services in parallel and waiting for all to complete, the bandwidth requirement is the sum of all individual bandwidth requirements while the time to complete is the maximum of their individual times to complete. Net cost on the other hand is the sum of all individual costs in *both* cases. It is clear that different measurements behave differently under different composition operations. This gets more complicated when conditional execution (such as failover) enters the picture. For conditional execution, one measurement (success rate, say) can affect all others if it is one of the variables in the condition. For example, if a backup service is executed just when the primary service fails, then all measurements need to take into account the serial execution of the backup service which occurs at some measurable rate.

1.2 Experimenting with composition properties

In all but the most trivial of examples, the effects composition has on the performance of a system are impossible to predict without calculation. These predictions are very important in answering questions such as:

- Which components should be optimised in order to reduce the overall time to complete?
- Which third party components should we attempt to reduce the cost of in order to reduce overall cost?
- Which components should we focus on debugging in order to improve overall reliability?

- What effect would reducing the cost of operation X have on the overall cost?
- Which component is causing us to breach our constraints?

We have implemented an engine that allows predictions to be made about composite service characteristics based upon the characteristics of the individual components. This system is detailed in Section 2.

In addition to being able to make such predictions it should be possible to experiment interactively with different configurations. Answering questions such as "Which components should be optimised in order to reduce the overall time to complete?" should not require the user to write a full specification for optimised versions of each component, in order to see how they perform. It should be possible to quickly change parameters of theoretical components and re-calculate predictions of composite service metrics in order to see what effect the change would have. Only with this experimental tuning of the properties of services would it be possible to answer these questions quickly and easily.

In QoS Explorer we have developed such a tool for experimenting with the properties of the services that make up a composition; we describe it in more detail in Section 3.

2 Inferring composite service properties

In order to explore the space of service choices, service characteristics and workflow modifications, it is necessary to calculate the effect that varying each has on the whole. Consequently we have developed **Agrajag**, a tool which calculates the aggregate behaviour of a composition in terms of its workflow structure and the known behaviour of its individual services. Applications built on top of this tool can take user specifications of workflows and choices of services, evaluate them with Agrajag, and present the user with a representation of how their composition might be expected to behave, allowing the user to develop a feel for which parts of their workflow are critical to the performance characteristics they're interested in and constraints against which they're operating.

As explained in the introduction, Agrajag starts with a workflow W, the service instances from which it is constructed $S_1 \ldots S_n$, and the known performance of each of those services P_{S_i}. It then computes the aggregate performance of the whole workflow, P_W:

$$\text{Agrajag}(W_{S_1 \ldots S_n}, P_{S_1}, \ldots P_{S_n}) \longmapsto P_{W_{S_1 \ldots S_n}}$$

While so sophisticated a representation of composition structure as WS-BPEL, or a scripting language, may not be necessary, at least the basics of parallel, conditional and serial operation need to be modelled. Parallel operations range from simple variations like waiting for the first or all responses through to more complicated ones such as voting or waiting for a certain number of responses to satisfy a condition (such as succeeding rather than failing).

From Agrajag's point of view, a service instance within a workflow is nothing more than a bundle of named typed measurements representing the known behaviour of that service[1]. These are the objects with which aggregation computations are made. It is necessary to process all measurements at once in these bundles because, once elements such as conditional operations and "parallel-first-response" are admitted to the workflow, individual measurements start to affect the probabilities with which other service instances are executed, hence affecting *all* measurements for dependent elements of the workflow.

The "measurements" mentioned in the last few paragraphs correspond to the QoS parameters in which the user is interested — things like time to complete, availability, accuracy, peak bandwidth usage, perhaps even cost. These actually each comprise three things: the *name*, analogous to a variable name in conventional programming; the *behaviour* (analogous to *type*), describing how the measurement will behave under the different composition elements; and the *value*, expressing the actual measurements for the service instance in question. *Behaviour* is extracted from a reference into our ontology, QoSOnt, as a mapping from workflow operations to (numerical or logical) operations on values. All workflow operations must eventually be applied to actual measurement *values* if useful predictions are to be made; the choice of underlying representation for these values will be critical, as (for example) it will be very difficult to say anything about the slowest 10% of calls to a service if all that's known about that service is its average performance. Possible representations for numeric quantities include expected value; minimum-maximum range; mean and variance of best normally distributed approximation; probability density function; Markov model of states; etc. Even within an individual problem it will almost certainly make sense to use different value representations for different properties. At the moment the main value representation used is an approximation of the measurement's statistical distribution (probability density function) using a variable number of uniform segments. This allows for moderately accurate representations of a number of situations, including delta functions, bimodal distributions, etc. Agrajag also supports values as probabilistic named states — e.g. a "completion" state might take the value "succeeded" 99.99% of the time, and "failed" 0.01% of the time; this succeeded-or-failed value being used to describe processes

[1] For our purposes, a "service instance" represents a single action (method call) of an actual service; a real web service might offer several such operations, but each operation needs to be represented as a separate entity within the workflow for accurate modelling.

which occasionally fail for unknown reasons, and being used to calculate the rate at which a backup service will be invoked.

To illustrate this, Table 1 presents evaluations by Agrajag of six different example workflows in the context of having two available implementations of a service X: a "premium" version with low probability of failure on demand (**PFD**) and good response time[2] (but high cost), and an "economy" version which is cheap but slow and prone to failure. The table details these three characteristics (time to complete (as a probability density function), probability of failure on demand, and cost) for all six choices.

The first two choices are straightforward: either choose the premium service (**A**) or the economy service (**B**) — here we see clearly how the first offers "four nines" and good performance at a cost while the other relinquishes these qualities in exchange for a cheaper service.

Next we have two failover structures, differentiated by the primary choice of service: using the premium service first (**C**) results in net behaviour dominated by that service's characteristics because it fails so infrequently — although the net probability of failure on demand has now dropped by one order of magnitude (assuming independence of failures among the services); using the economy service first (**D**) however results in an interesting blend of characteristics — the same low failure rate as seen in **C** but at a low cost, albeit with fairly poor performance. Note that for these two structures cost is bimodal: for **D**, X_{economy} succeeds 90% of the time at a cost of £1, but 10% of the time its failure results in fallback to X_{premium} at a subsequent net cost of £11. Thus over repeated use a low average cost of £2 is observed, but for any individual use there is a 10% risk of a much higher £11 cost. Regular users might be prepared to take this risk, amortising the occasional expensive situation over many cheap ones, while infrequent users might be uncomfortable at the prospect of a significant chance of paying five times the average price.

The final pair of choices are based around another structure — invoking *both* services at the same time[3], immediately using the first response in the case of **E**, and waiting for both responses and comparing them in **F**. **E** consequently offers the best performance of all choices, at a high cost and with low probability of failure (Gashi et al. [11, 10] have demonstrated substantial performance increases using this model with diversely implemented databases). **F** on the other hand seems to have no merit whatsoever — highest cost, highest failure rate, and worst performance. This is however entirely a consequence of what we have chosen to model: if we also consider the likelihood of a *correct* result, then **F** would be the best performer — while it cannot repair byzantine failures, it can at least identify situations where we are unsure of the result and improve confidence in the results it asserts to be correct, although it achieves this by introducing a new failure mode and increasing the failure rate (and can only even do this where failures are independent and diverse).

[2]The "lump" on its time graph is characteristic of an internal timeout-based back-up within the service, and demonstrates the usefulness of Agrajag's more detailed distribution-based models.

[3]Note that for certain services, most notably those with side effects, this kind of structure is *not* acceptable.

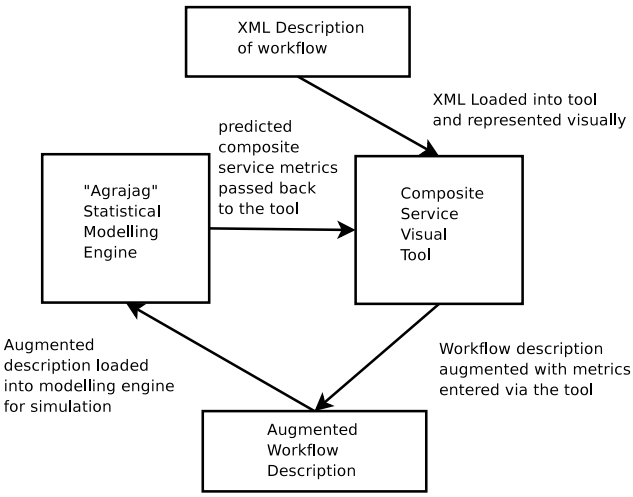


Figure 1. How QoS explorer works

Of particular use to systems designers is the fact that the objects that Agrajag generates (representing particular compositions of particular services) support rich statistical queries such as "What is the maximum cost of this workflow?", "Will no more than 10% of responses take longer than 5s?", "Is the average response time less than 2.5s?", etc.

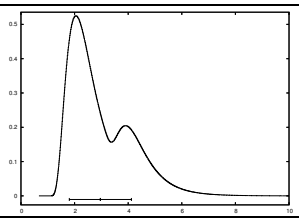
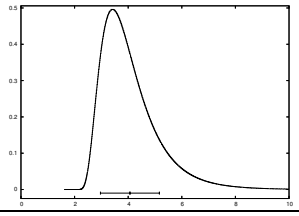
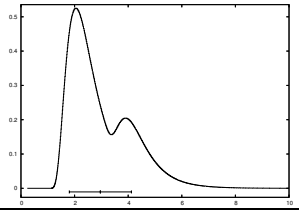
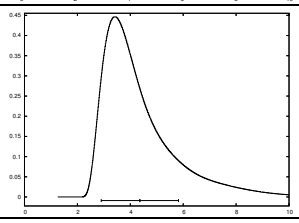
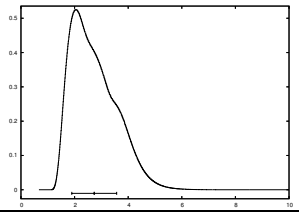
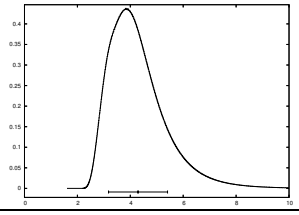
3 QoS Explorer: a tool for experimenting with composite service metrics

QoS Explorer is a software tool we have developed for predicting composite service metrics based upon those of a workflow's constituents. The tool allows the properties of components within the workflow to be altered manually in order to experiment with different configurations and to optimise overall performance.

The architecture of QoS Explorer and the main interactions between the graphical component and the modelling engine are shown in Figure 1.

QoS Explorer reads in an XML representation of the workflow and renders a visual representation, such as that shown in Figure 2. The fragment of the workflow shown in Figure 2 consists of services in series and parallel. The flow of control is from top to bottom. Service C would be executed, followed by Services 1, 2 and 3 in parallel, followed

Table 1. Effect of choice and structure on behaviour. The small bars under the "time to complete" curves represent $[\mu - \sigma, \mu, \mu + \sigma]$.

Workflow	Time to complete (seconds)	PFD	Cost (£)
Start → X_{premium} (succeed → Done; fail) **A**		0.00010	10
Start → X_{economy} (succeed → Done; fail) **B**		0.10000	1
Start → X_{premium} (succeed → Done; fail → X_{economy} (succeed → Done; fail)) **C**		0.00001	$\mu = 10.0001$ 99.99% 10 0.01% 11
Start → X_{economy} (succeed → Done; fail → X_{premium} (succeed → Done; fail)) **D**		0.00001	$\mu = 2$ 90% 1 10% 11
Start → X_{premium} / X_{economy} (succeed / fail) → First (succeed → Done; fail) **E**		0.00001	11
Start → X_{premium} / X_{economy} (succeed / fail) → Both (succeed → Done; fail) **F**		0.10009	11

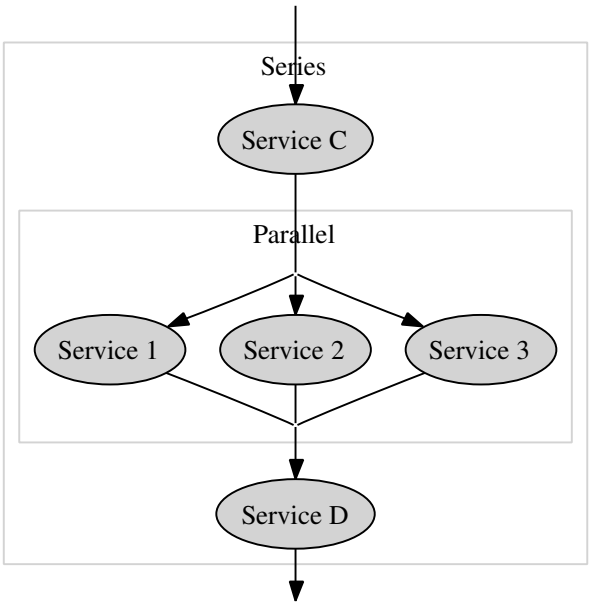


Figure 2. Graphical representation of a workflow

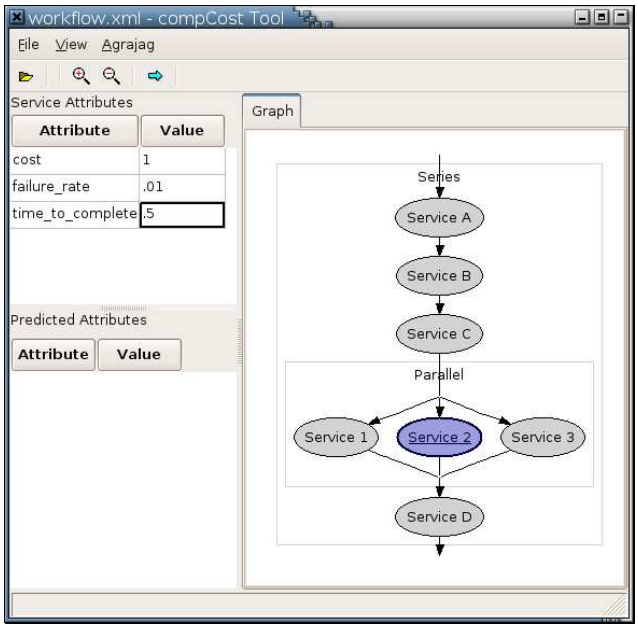


Figure 3. Entering metrics for a service

by Service D.

The QoS characteristics of a service are entered by first selecting the service by clicking on it, and then entering the relevant characteristics into the table that appears to the left of the graph, as shown in Figure 3.

The data entered can be actual data from measurement of the performance of existing components, data published by third parties or estimated data. Where the user is trying to determine which components are the bottleneck in a composition or where to focus optimisation in order to achieve the most benefit, repeated experimentation while varying characteristics of different elements of the workflow will help locate critical components.

When aggregate behaviour is calculated by clicking on the execute button in the toolbar, the original XML description of the workflow is augmented with the metrics provided to QoS Explorer by the user. This description of the workflow configuration is then passed to the Agrajag engine which generates and returns a document describing the aggregate performance of the workflow, as shown in Figure 1. This information is displayed in a table to the left of the the workflow graph, as can be seen in Figure 4.

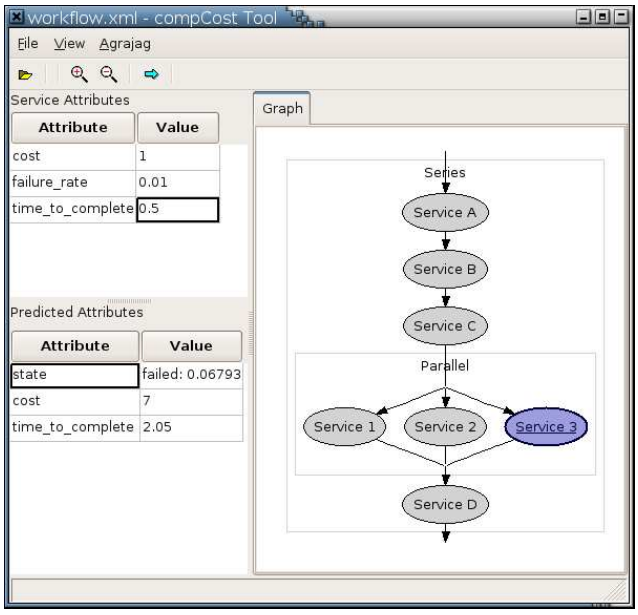


Figure 4. Predicted aggregate performance

3.1 Example

Figure 5 shows a slightly more complex workflow, one in which failure of part of the workflow will cause a backup component to be invoked. Anybody using QoS Explorer

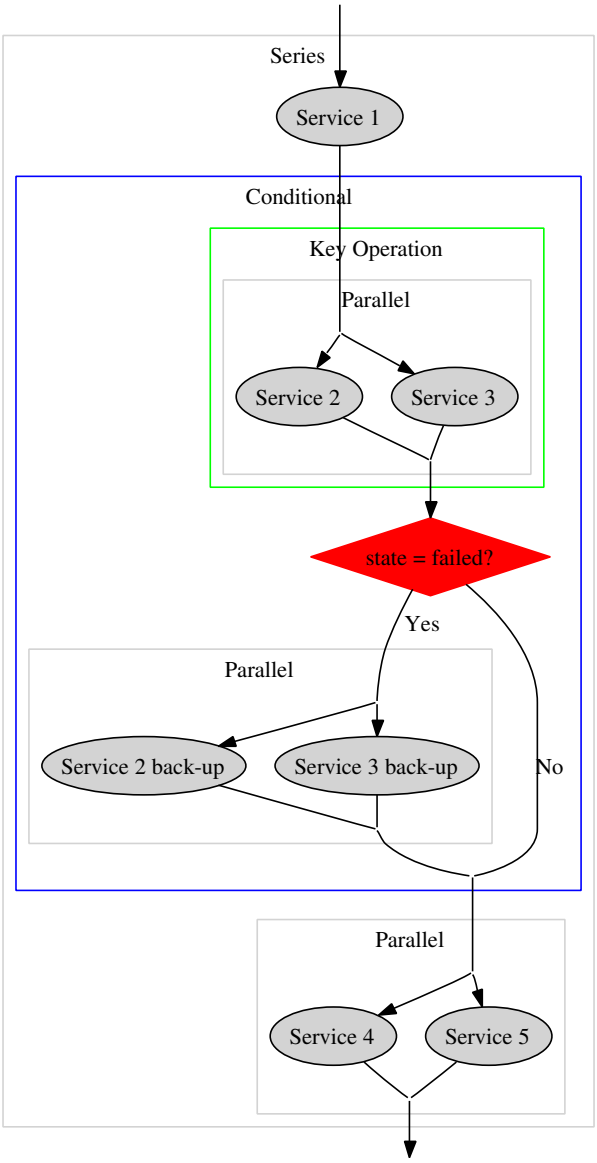


Figure 5. A workflow with a failover conditional

would quickly discover that work invested in (or withdrawn from) the back-up component will have little effect on many overall system characteristics, and consequently direct their investment to other parts of the workflow. While this is easy to see in an example with so few services, as workflows become larger and more complex it becomes much more difficult to develop an intuitive grasp of which elements are most critical to non-functional objectives. As this becomes the case we expect QoS Explorer will really show its worth, easing identification of the services with most impact on the user's objectives, and making clear those which have little impact on them — consequently allowing the workflow developer to focus work or investment only where it's needed.

4 Conclusions and further work

QoS Explorer allows anyone building a composite web service to predict the behaviour of the system they are assembling in terms of a range of metrics, based upon the properties of the services composed. In addition to predicting the values of metrics, the tool can be used interactively to determine which services in a composition are most important to the overall performance of the system in terms of a given metric. This can be particularly useful where there are constraints that must be met as it allows the user to ensure that the services they select for particular parts of the workflow will allow them to meet these constraints.

While copious work has been done recently in this area [14, 2, 19], much of it is predicated on the existence of a competitive market in compatible services, as well as mechanisms for automatic negotiation and entry into contracts with the providers of such services. Such a situation is not yet the case, and QoS Explorer addresses the *real* current situation of composition development and service contract negotiation still being a largely human-centered practice.

"Under the hood," the prediction of the performance of the composite service is carried out by Agrajag, a probability-density-function-based statistical modelling engine we have built. This technique is moderately compute-intensive but is more accurate and allows far richer queries to be made against workflow performance than would be possible with other techniques (existing work seems to exclusively use numeric constants as variables). The validity and benefits of abstracting out a basic set of "workflow patterns" instead of tying yourself to one complete workflow language have been observed by Jaeger et al. [13] among others.

QoS Explorer can be used to get a reliable estimate of the performance of a composite service, where this information would otherwise not be available or would be costly to acquire.

The workflow operators supported by the tool are cur-

rently limited to a subset of the most commonly used, but the system is extensible, allowing new operators to be implemented in a modular fashion. QoS Explorer also does not allow recursion in workflows for more fundamental reasons, but in practice this isn't often necessary. Also, this work assumes independence of services, variables, failures; it might be possible to apply research which quantifies the level of (in)dependence in certain situations (such as [22]) to offer more realistic results.

In the future we want to add a number of metrics to the core modelling engine, in order to allow prediction of an even larger range of characteristics. Plans for the graphical component of the tool include improving ease of entry of probability distribution parameters. The workflow description could be used to find a range of candidate services from a directory, and the metrics used in aggregate calculations could be obtained from descriptions of these services. Automation of some tasks, using the more machine optimisation-focused ideas in [5, 18, 23] while drawing lessons from Jaeger's et al. [12] evaluation of selection algorithms also seems desirable, so that QoS Explorer can offer directed service choice recommendations to the user, and highlight workflow bottlenecks and critical paths automatically.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[2] D. Ardagna and B. Pernici. Global and local QoS constraints guarantee in web service selection. In *IEEE International Conference on Web Services (ICWS'05)*, pages 805–806. IEEE, July 2005.

[3] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. Web service choreography interface. http://www.w3.org/TR/wsci/, 2002.

[4] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. Web services conversation language. http://www.w3.org/TR/wscl10/, 2002.

[5] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. QoS-aware replanning of composite web services. In *IEEE International Conference on Web Services (ICWS'05)*, pages 121–129. IEEE, July 2005.

[6] M. Corporation. XLANG — web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[7] G. Dobson, R. Lock, and I. Sommerville. QoSOnt: a QoS ontology for service-centric systems. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 80–87, 2005.

[8] I. F. Leymann. Web services flow language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, 2001.

[9] S. Frolund and J. Koisten. QML: A language for quality of service specification. http://www.hpl.hp.com/techreports/98/HPL-98-10.html, 1998.

[10] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. On designing dependable services with diverse off-the-shelf SQL servers. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, Lecture Notes in Computer Science, pages 191–214. Springer-Verlag, 2004.

[11] I. Gashi, P. Popov, and L. Strigini. Fault diversity among off-the-shelf SQL database servers. In *Proc. DSN 2004*, pages 389–398, 2004. International Conference on Dependable Systems and Networks, Florence, Italy.

[12] M. C. Jaeger, G. Muhl, and S. Golze. QoS-aware composition of web services: A look at selection algorithms. In *IEEE International Conference on Web Services (ICWS'05)*, pages 807–808. IEEE, July 2005.

[13] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS aggregation for web service composition using workflow patterns. In *Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 149–159. IEEE, 2004.

[14] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS aggregation in web service compositions. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, pages 181–185. IEEE Computer Society, 2005.

[15] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language. http://www.w3.org/TR/ws-cdl-10/, 2004.

[16] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *IBM Research Report*, May 2002.

[17] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A workflow framework for grid services, 2002. In Preprint ANL/MCS-P980-0802, Argonne National Laboratory.

[18] Y. Li, K. Sun, J. Qiu, and Y. Chen. Self-reconfiguration of service-based systems: A case study for service level agreements and resource optimization. In *IEEE International Conference on Web Services (ICWS'05)*, pages 266–273. IEEE, July 2005.

[19] D. A. Menasce. Composing web services: A QoS view. *IEEE Internet Computing*, 08(6):88–90, November/December 2004.

[20] OASIS. Web services business process execution language version 2.0 (committee draft). http://www.oasis-open.org/apps/org/workgroup/wsbpel/, 2005.

[21] W. M. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[22] M. van der Meulen and M. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In M. D. Cin, M. Kaaniche, and A. Pataricza, editors, *5th European Dependable Computing Conference (EDDC-5)*, Lecture Notes in Computer Science, pages 199–209, Budapest, Hungary, April 2005. Springer-Verlag.

[23] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 411–421. ACM Press, 2003.