Experiments Towards Adaptation of Concurrent Workflows

J. Smith, P. Watson.

TECHNICAL REPORT SERIES

No. CS-TR-1039 July, 2007

TECHNICAL REPORT SERIES

No. CS-TR-1039 **July, 2007**

Experiments Towards Adaptation of Concurrent Workflows

Jim Smith, Paul Watson.

Abstract

This paper is concerned with the adaptive execution of workflows on a resources consisting of a pool of machines and a pool of alternative web services. The hierarchical nature of workflows enables adaptation at multiple levels. In this work, adaptivity is concerned with changing the mapping of services to machines and workflow invocations to services, in order to meet the requirements of both user and provider. Specifically, a third-party workflow engine (ActiveBPEL) has been wrapped to support mapping at these two levels. Results are presented for experiments within a cluster of machines which demonstrate a benefit from adapting in response to changes of user load and to changes in the pool of alternative services available during a workload. The experiments include a range of adaptivity scenarios and show that, by selection of an appropriate policy, a significant gain can be made.

Bibliographical details

SMITH, J., WATSON, P.

Experiments Towards Adaptation of Concurrent Workflows
[By] J. Smith, P. Watson.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2007.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1039)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1039

Abstract

This paper is concerned with the adaptive execution of workflows on a resources consisting of a pool of machines and a pool of alternative web services. The hierarchical nature of workflows enables adaptation at multiple levels. In this work, adaptivity is concerned with changing the mapping of services to machines and workflow invocations to services, in order to meet the requirements of both user and provider. Specifically, a third-party workflow engine (ActiveBPEL) has been wrapped to support mapping at these two levels. Results are presented for experiments within a cluster of machines which demonstrate a benefit from adapting in response to changes of user load and to changes in the pool of alternative services available during a workload. The experiments include a range of adaptivity scenarios and show that, by selection of an appropriate policy, a significant gain can be made.

About the author

Jim Smith worked as a computer programmer for some years with the UK electricity supply industry, then moved to the University of Newcastle upon Tyne where, following studies for MSC and PHD, he is working as an RA on the Polar* project.

Paul Watson is Professor of Computer Science and Director of the North East Regional e-Science Centre. He graduated in 1983 with a BSc (I) in Computer Engineering from Manchester University, followed by a PhD in 1986. In the 80s, as a Lecturer at Manchester University, he was a designer of the Alvey Flagship and Esprit EDS systems. From 1990-5 he worked for ICL as a system designer of the Goldrush MegaServer parallel database server, which was released as a product in 1994. In August 1995 he moved to Newcastle University, where he has been an investigator on research projects worth over £13M. His research interests are in scalable information management. This includes parallel database servers, data-intensive e-science and grid computing. In total, he has over thirty refereed publications, and three patents. Professor Watson is a Chartered Engineer, a Fellow of the British Computer Society, and a member of the UK Computing Research Committee.

Suggested keywords

ADAPTIVE EXECUTION,
BPEL,
DYNAMIC DEPLOYMENT,
EXPERIMENT,
QOS BASED EXECUTION

Experiments Towards Adaptation of Concurrent Workflows

Jim Smith and Paul Watson

***Abstract*—**
This paper is concerned with the adaptive execution of workflows on a resources consisting of a pool of machines and a pool of alternative web services. The hierarchical nature of workflows enables adaptation at multiple levels. In this work, adaptivity is concerned with changing the mapping of services to machines and workflow invocations to services, in order to meet the requirements of both user and provider. Specifically, a third-party workflow engine (ActiveBPEL) has been wrapped to support mapping at these two levels. Results are presented for experiments within a cluster of machines which demonstrate a benefit from adapting in response to changes of user load and to changes in the pool of alternative services available during a workload. The experiments include a range of adaptivity scenarios and show that, by selection of an appropriate policy, a significant gain can be made.

I. Introduction

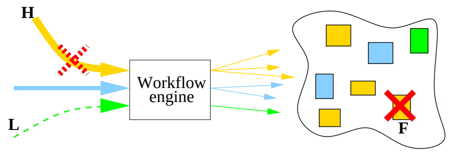
The increasing use of outsourcing has prompted the development of standard templates for the legal contracts, known as Service Level Agreements (SLA) which define the relationship between service provider and user. An SLA typically defines obligations, costs, monitoring and penalties. While these contracts originated as paper documents, the appropriateness to the computer based out-sourcing facilitated by web services, and the potential for automation, has been recognized for some time, [21], [6]. While enforcement of SLAs has tended to be more of an issue in commercial settings [15], there is increasing impetus towards the adoption of SLAs in scientific contexts, e.g. [7], where service provisioning has traditionally been best-effort, but where a greater emphasis is placed on dynamic collaborations, and where perhaps also complex, one-off requests entailing access to widely distributed and large scale resources are more common. Indeed, it would not be surprising for an SLA in a scientific grid setting to be set up for a single job. Accordingly, there is work which seeks to dynamically replan complex distributed workflows in order to best satisfy non-functional requirements [5], as might be specified in an SLA. There remains an issue as to how best to arbitrate resources between concurrent users.

The presence of an SLA gives assurance to a client that if, for instance, they keep to the agreed limits regarding service requests and resource usage, the provider will meet agreed requirements, or incur an agreed penalty. It also allows the provider to plan resource provision in the guarantee of a certain defined usage. The provider can plan to share a collection of resources across the anticipated set of user workloads. If a client is very sure of a precise and constant requirement for the agreed period, the SLA will be a close fit. If all clients have these characteristics, the resources will be well used under a regime of static matching. In practice, an SLA inevitably offers some freedom on both sides. The discrete nature of monitoring permits a degree of freedom to the client, which the provider is obliged to allow for. If the agreement permits a specific average request rate and monitoring interval, the actual request rate can be quite "bursty" within those constraints. If the provider guarantees a certain average response time (or perhaps a certain average result accuracy), then the resulting attribute for a particular request can still be quite unsatisfactory. If the provider sets aside resources to meet the peaks in a workload, these resources may be underused much of the time. It is possible to address variability of load applied to web services from a modelling perspective and derive optimal strategies [18]. The focus here is aimed at using experimentation to explore a more general case, and is more pragmatic.

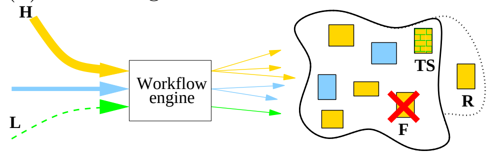
Owning a pass for a car park yet finding no space, or being transferred to a "different flight" from one's booking may be seen as suggesting that a penalty is not seen just as a last resort, but more as a calculated risk, as resources are matched more and more tightly to requirements. If the requirements for the period of an SLA are not predicted with great accuracy, a user may yet submit a load whose profile falls outside of the current agreement. Similarly, a provider may deliberately choose to default on a particular agreement; preferably one with low penalty. While such behaviour seems inevitable, assuming that workload will tend to outstrip resources, there is clearly an advantage to being more flexible in meeting SLAs. This flexibility will not just be attractive to the user, but must also benefit the provider, since better resource utilization can be achieved. The work described here aims ultimately to build infrastructure that offers support for dynamic management of varying workloads where individual requests are complex and variable and have associated non-functional requirements. Specifically, this work seeks to add practical support for adaptivity to a static workflow engine.

For example, suppose static and adaptive engines support three concurrent workloads as illustrated in figure 1. In this scenario, **H** is a heavy load and **L** is a very light and intermittent load. While the static engine must allocate resources in accordance with the agreed SLA, an adaptive engine can time-share a single resource **TS** dynamically between two separate users, here **H** and **L**, thereby reducing over-provisioning. In the event that a resource **F** fails, a static engine defaults on the corresponding SLA, which might incur a significant financial penalty. By contrast, an adaptive engine can obtain a replacement resource, here **R**, to replace one that has failed, or by reallocating resources, possibly default on a less profitable SLA.

This paper describes how an existing workflow engine can be wrapped with some additional infrastructure in-



(a) static engine



(b) adaptive engine

Fig. 1. Comparing static and adaptive workflow engines.

cluding a dynamic service deployment utility and a small amount of "glue" to implement an adaptive workflow engine that has support for adaptivity at multiple levels. Separating out the adaptivity support from the re-used workflow engine clearly has limitations, but there is also a great saving in building on the established component. Thus, it is possible already to present measured responses of a first prototype in a number of different scenarios of changing workload. In addition, there is a potential for applying the wrapper to alternative basic workflow engines.

The remainder of the paper is structured as follows. Section II discusses related work. Section III outlines how support for adaptivity can be added by wrapping an existing workflow engine. Section IV mentions early work on a prototype adaptive engine. Section V presents some experimental results, obtained using this prototype. Finally, section VI concludes.

II. Related Work

There are many tools which support the construction of composite services, or workflows, from published component services, but they may be classified according to the degree to which workflow composition and mapping is automated:

functional semantic There are systems which aim to exploit functional semantic information to construct a workflow from an arbitrary pool of component services according to some high level description. In restricted scenarios where the composition features request-response style orchestrations, systems have been presented which for instance use rule-based processing [19] or planning [5]. Work towards composition where the interactions are more complex, e.g. asynchronous is on-going [12].

non-functional At another level there are systems where the user specifies a functional composition explicitly, but where each invocation can be mapped to one of a set of functionally equivalent alternative services according to non-functional requirements dynamically at run-time [25], [1], [24].

physical resource At a lower level still, a workflow defines explicitly not just the macro-level control flow, but also the specific task/service alternatives that are invoked. A mapping to physical resources is then performed by the infrastructure. Examples of such support occur in workflow systems targeting scientific applications, e.g. [10]. This support is less common in business scenarios, where applications are more likely to make updates to persistent data; issues in this area are described in [4].

manual At the lowest level, a workflow definition specifies particular resources, i.e. particular physical machines and/or service instances. Application workflows are commonly expressed at this level in practice.

Ideally, an adaptive engine would support automatic mapping at the highest level. There are projects which seek ultimately to achieve this. However, the emphasis appears to be on the mapping and re-mapping techniques themselves, and to focus on meeting requirements placed by a user on a particular execution, e.g. to minimize response time. By contrast the work described here seeks to investigate the use of such mapping and re-mapping techniques in order to manipulate concurrent workflow executions in the context of requirements expressed by both user and provider. The scenario envisaged in the current work is of an organization providing for the execution of workflows which invoke services from a pool maintained by the organization, over a pool of resources also maintained by the organization, in the manner of autonomic service provision, [14]. There is considerable research of this nature in the context of application hosting systems, e.g. [2], [22]. While recent techniques are becoming scalable, such systems tend to assume that the applications being managed are of simple, i.e. not composite, structure. In the context of composite services, [20] presents results of a simulation study showing a potential for benefit through varying the degree of replication of services invoked by a composite service. Such adaptations can be seen as being associated with the *physical resource* level. By contrast, the work described here includes service mapping adaptations at the *non-functional* level, which can also be beneficial. in addition, the work described here presents results from experiments in a real cluster rather than a simulation.

With a view to exploiting readily available technology, the work is set in the context of BPEL [17], which is well established in commercial settings and for which there are available robust workflow engines that scale well under concurrent workloads. There is other work on supporting adaptivity in BPEL. For instance [13] describes an enhancement to the language which allows specification of non-functional requirements that can be used to guide a selection between alternative semantically equivalent services. The enhanced invocation is implemented directly in the open source ActiveBPEL [8] engine. Since the engine itself is still now under development, maintaining an enhancement separately from the main development team is inevitably a problem. This issue is addressed in [9], where plans to use Aspect Oriented techniques are outlined. However, in neither case are there any experimental measurements of complete adaptation scenarios. The work described here aims to avoid the engineering issues entailed in modifying the third-party engine itself. The language extension described by [13] could be used here, but the approach would be to extract the additional non-functional information for invocations made in a BPEL program and pass it to a separate service mapping component. Other recent work [3] is also concerned with adapting BPEL workflows during execution, for instance to select between available alternatives on the basis of non-functional requirements, or to try an alternative service on failure of the first choice. In common with the work described here, [3] uses a third party BPEL engine unchanged, but unlike the work described here, it focusses on the user perspective.

Overall, the distinctive features of the work described here are: that it considers adaptations from the viewpoints of both user and provider; that it enables adaptations of both service invocation and physical resource mapping; that it re-uses a basic workflow engine without alteration; and that it presents measured results demonstrating example feedback loops, obtained from a practical prototype running in a cluster environment.

III. An approach to making workflow management adaptive

The first step is to identify the requirements for adaptivity, and therefore the key variables. The next step is to identify mechanisms within a workflow engine that may be manipulated to control those variables. Finally, measurements are identified which can be used to guide these controls.

As described earlier, the focus for this work is on mapping invocations within workflows to alternative services in respect of non-functional requirements, and on mapping services to machines. To this end, the adaptive workflow engine envisaged here has access to both a pool of services and a pool of computational resources and can freely deploy instances of those services onto computational resources in order to suit workload requirements. In addition there are alternatives amongst the services, which can be distinguished by non-functional properties, such as *monetary cost*, *result precision* etc. The aim ultimately is to be able to dynamically map workflow invocations onto service alternatives, and thence to physical resources so as to support SLAs such as 'maximize result precision while keeping average response time and average monetary cost below specified limits'. One mechanism is clearly that of mapping invocations to services. Another is created by dividing the available pool of computational resources into subpools and mapping each workload into a separate subpool, whose size is then varied so as to accelerate or decelerate that workload.

Three controls on workflow execution are defined in the system described here.

subpool size Control of the size of a subpool can be used to influence the overall throughput of a workload - and average response times for individual requests.

invocation mapping Any invocation can be mapped dynamically to any of the alternatives within its mapping set in order to select particular non-functional properties. Such a mapping set can vary dynamically as services are added/removed.

resource mapping Within a subpool load can be directed more or less to specific resources, for instance to allow for resource heterogeneity, or external load.

These controls act at different levels, of subpool, invocation and resource. However, there is some overlap in effect. For instance, it is possible to reduce workload response time by increasing subpool size or by mapping all (or even just one) invocation(s) to faster alternative service(s). The distinction is that changing invocation mapping typically affects non-functional properties other than response time, e.g. result quality.

In support of these controls, a number of measurements may be made, including those listed below.

internal measurements offer, in principle, access to arbitrary internal attributes of the workflow engine; for instance to internal queue sizes. It is thus easily possible to find information regarding the progress of requests in the system, and very cheaply. This is powerful, but implies intrusion into the workings of the engine and is therefore not portable across diverse engines. There is also a development overhead in navigating through the internals of an engine. As described earlier the implementation strategy employed in this particular work specifically avoids this approach.

workflow requests can be recorded in order to compute the request rate and response times. Assuming each request identifies user and workflow, it is possible to compute these metrics per workload.

service invocations can be measured to compute the rate and response times of invocations to particular service instances. Assuming that the request identifies not only user and workflow but also which component invocation the call corresponds to (i.e. which node in the workflow graph) and which particular service is being invoked, measurements here can identify request rates and response times at the granularity of individual alternate services. In general, an invoked service could itself be composite. Of course, in that case, the measurement that counts service invocations from the point of view of the parent service counts workflow requests from the point of the child service.

direct resource measurements may be employed to access attributes of a particular computational resource, e.g. CPU or memory availability, or service running on that resource.

indirect resource measurements are recorded separately from the requesting engine and stored for subsequent access by the engine, possibly amongst other applications. Two classes of registry are anticipated. Administered as a part of the adaptive workflow engine, a local engine might be used to store attributes such as monetary cost and result precision. Alternatively, attributes which might rely on update by community or trusted authority, such as reputation or security attributes may be best served within a registry that is administered separately from the workflow engine. However, for the adaptivity infrastructure which is using these attributes, the immediate significance seems likely to be just the need to access different registries for different attributes.

Figure 2, illustrates mechanisms described above. The

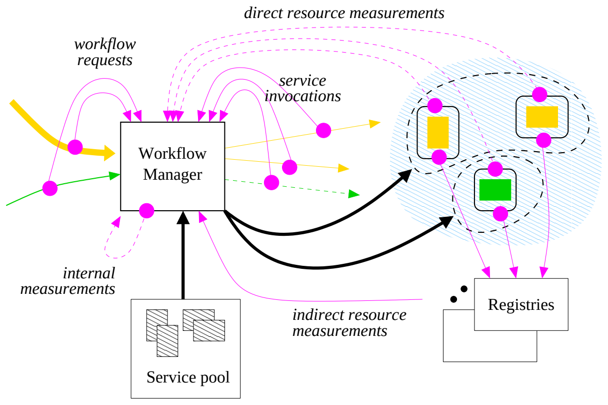


Fig. 2. Controls in an adaptive workflow engine.

current prototype, which is described in the following section, implements the two controls (highlighted), *subpool size* and *invocation mapping*. It exploits measurements of *workflow requests* and *service invocations* and accesses non-functional properties stored with service definitions in a local registry through *indirect resource measurements*.

IV. Implementation

An implementation of the support for adaptivity described in section III is ongoing. As described earlier, this work aims to use the basic workflow management support in “black-box” fashion. While it proves possible to avoid manipulating the internals of the basic engine, it is necessary for some of the implementation to be specific to the workflow language supported by the basic engine. One reason for this is that, as described in the previous section, it is necessary to ensure that suitable identifying tags are present in service invocations. However, it is also necessary for the adaptivity support which wraps the basic engine has to have some understanding of the structure of workflows while they are being executed, in order to make policy decisions. These issues are further described later in the section, but essentially it is necessary to:

- parse a submitted workflow in order to generate restricted descriptions of its structure for use by the adaptivity infrastructure;
- make simple edits to the submitted workflow to insert identification tags into invocations, as additional parameters.

Figure 3 shows how a level of support for adaptivity has been added to an existing workflow engine. The controls implemented in this structure are at the three levels described, of PoolAmin, MapAdmin, and LoadBalance. The figure also where functionality is not fully implemented, by dashed lines. The flow of user requests against work-

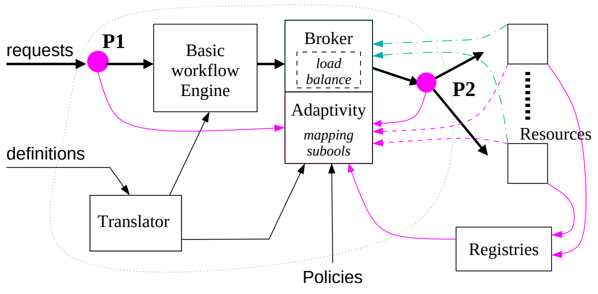


Fig. 3. A possible structure of an adaptive workflow engine

flows and thence to individual service invocations is shown by the thick arrows. While the basic workflow engine is responsible for executing concurrent workflows, all invocations are intercepted by a broker which supports parallelization of streams of requests over multiple resources. Probes **P1** and **P2**, which actually share a common implementation, intercept workflow requests and service invocations to supply the measurements required by the adaptivity component that is a loosely coupled extension of the broker.

It is common practice for workflow execution to be a two step process. The first entails installing a workflow schema into the engine; only after completion of this step can execution of the workflow be requested. For the adaptive engine described here, a translator component processes any new workflow in order, as mentioned earlier, both to insert identification tags required for monitoring and processing within the adaptivity component, e.g. mapping and pool selection, and to generate data structures describing required aspects of the workflow structure which are used by the adaptivity. The remainder of this section gives some discussion of the components which are additional to the basic workflow engine

A. Broker

The publicly available Dynasoar system [23] implements redirection of incoming service invocations to service instances deployed dynamically on a configured collection of hardware resources as required. Dynasoar maintains a registry describing services which have been registered to it and a store of deployable services. Currently Dynasoar distributes invocations statically across the resources available to it, but does queue requests internally, at any time issuing a number of requests so as to occupy the number of CPUs available on a machine, but not so as to flood the machine with requests. Ongoing work in the authors' institution is concerned with extending Dynasoar to incorporate measurements of attributes, e.g. memory and CPU availability, of resources it deploys to in order to implement load balancing. The experiments described later rely on the static provision. Dynasoar implements three services as described below.

- Service Provider (SP) implements the public interface for any service which is managed by Dynasoar. For a registered service, a user can submit a request to the corresponding SP, which takes care of forwarding that request.

- Host Provider (HP) acts as interface to one or more physical resources. Typically HPs are configured into a hierarchy, where a LEAF level HP interfaces a single machine while an intermediate HP acts as interface to one or more clusters of machines, and encapsulates a policy for scheduling requests to the lower level HPs it manages. A ROOT HP, which typically interfaces a single cluster of machines, is ideally placed to implement load-balancing within its cluster.
- Code Store (CS) implements a repository for services which are managed by Dynasoar. This comprises a simple file store which contains a war-file for each registered service and a UDDI based registry where describing attributes and a cross reference to the executable can be stored.

B. Registry

The data required to support service invocation mapping is stored in the same UDDI based registry which is used already by Dynasoar. This is a Grimoires implementation [11], which supports version 2 of UDDI specification, but adds some extensions to support annotation with metadata. Services which are registered to Dynasoar are represented in the registry and those representations are annotated with a link to the WSDL definition. The registry also supports a service being annotated with non-functional properties, e.g. monetary cost. A search through the registry for all services which implement the WSDL corresponding to a particular invocation within a workflow can be employed in support of service mapping. Having found multiple alternate services, the annotations defining non-functional properties can be retrieved to permit a choice between these according to user requirements. One issue that arises is the cost of accessing the registry. In the experiments described here, polling is satisfactory, but in a realistic scenario, regular polling to access data corresponding to many alternative services for many workflows might represent a significant cost. In this context, support for callbacks seems appropriate, as supported by the subscription based access defined in version 3 of the UDDI specification [16].

C. Translator

The translator allocates to the workflow a unique number and to each node (corresponding to an invocation) a number which is unique within the workflow. These are referred to as *workflowID* and *invocationID* respectively. In this work, adaptivity support is concerned with relationships amongst invocations and between invocations and workflows; and not currently with other details of the workflow. Thus, the translator creates an XML file which documents the graph structure of a workflow, annotated with workflowID, invocationID and the identifying key, within the registry, of the WSDL definition that corresponds to each link. This XML file is passed to the adaptivity component to support service mapping. In addition, the translator generates a modified workflow definition which inserts the appropriate workflowID and invocationID as extra parameters into each invocation. As part of its handling of a request, the adaptivity component relocates these values to a message header, so that the end service invocation has the correct number of parameters. In addition to the workflowID, it is also necessary to obtain from a workflow request, and propagate, the userID, since it is the combination of the two which identifies a workload. A pragmatic solution for experimental purposes is simply to assume that the userID will appear as a parameter to the workflow. The Probe can the access the value and translated workflow can also propagate it in the same way as the workflowID and invocationID. This is not as elegant however as fetching the target operation name and obtaining from that the workflowID. However, it seems reasonable that in the context of a secured access to the workflow engine, the userID should be obtainable directly or indirectly via security headers. Since application workflow code seems unlikely to support access to header parts of messages, the idea would be for the probe to not only fetch the userID but also relocate it as an extra parameter to the translated workflow which can then propagate it as described.

D. Probe

By intercepting all messages directed through it, the probe maintains a small collection of statistics which it makes available via a publish-subscribe interface, in this case to the adaptivity component. Probe maintains a count of all such requests received, all of those that have completed and the sum of the durations of all such requests, i.e. in each case the time of completion minus the time of initiation. As well as these long term statistics, Probe also supports maintenance of similar statistics over a shorter interval, which may be specified in a subscription request. It achieves this by keeping an event list and computing short-term statistics when required for data reports. The statistics are organized into hash tables according to a key which is constructed from a set of values identified in the intercepted messages. In the case of **P1** for instance the operation name from the request body, identifies the workflow being requested. In the case of **P2** the assigned ID of the workflow is obtained from a field in a message header created by the adaptivity component.

E. Adaptivity

The adaptivity component is responsible for determining subpool size and invocation mapping. To do this, adaptivity subscribes to the two probes and thereby receives, incrementally, the statistics which it uses to decide on controls. In the current implementation, a number of policies, for setting subpool size and for selecting invocation mappings, are hard-coded in the Java implementation of the adaptivity component. A selection is made by hand at startup via a properties file. Dynasoar is modified slightly to invoke the adaptivity functions during processing of service invocations. Currently, adaptivity support is associated with a ROOT host provider (HP), typically one that interfaces a cluster - a single pool of resources. When a request message is first received its workload is first identified from the attributes contained in the message, e.g. userID, workflowID. This identifies a destination subpool. However, as described earlier, messages are released at a controlled rate. Specifically, a number of tickets is defined for each subpool, depending on the number of machines and processors. If all tickets allowed for the appropriate subpool are in use, the request is queued pending completion of one of the active requests. When a ticket becomes available, a request, e.g. the oldest, is dequeued and passed to adaptivity which implements the invocation mapping according to the current definition for the corresponding workflow. On return the request is forwarded to one of the machines in the current subpool, either directly to a previously deployed copy of the service, or to the local HP which will fetch a copy of the service from the repository for deployment and then forward the request to the newly deployed copy. Adding an extra machine to a subpool simply entails adding the machine to the set associated with the subpool, and increasing the number of tickets available for the subpool in accordance with the characteristics of the new machine. When a subpool size is reduced, the presence of active requests typically delays the implementation of that change. The target machine set size and target ticket allocation for the pool are altered immediately. If there are sufficient free tickets, they are collected immediately. Similarly, if there is a machine in the set for the subpool which is currently free of active requests, it is removed from the set immediately. Otherwise, tickets are collected as requests complete until the actual ticket allocation reaches the target ticket allocation. At that point the machine with the lowest number of active tickets is marked to receive no new tickets till the subpool mapping is resolved. This happens when sufficient machines in the pool have become free of active requests.

V. Experiment

A. Basic workflow management

In this work, the ActiveBpel [8] engine is employed as the basis for workflow execution. Interaction with ActiveBPEL is in the two phases described earlier as typical for workflow engines.

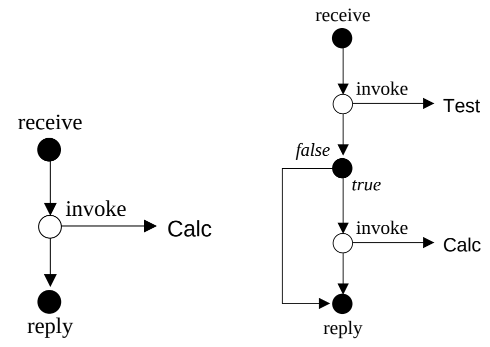
deployment In the first step a BPEL program is created and packaged into a jar file which includes necessary schema related files as well as the BPEL program itself. Deployment of this jar file is initiated by copying it into an appropriate directory managed by the engine, similar to the deployment of any web service. When the engine recognizes the new deployment it extracts the package and completes deployment, making the new workflow available as a service which may receive requests.

request Subsequently, a user may send messages to request execution of the registered workflow, supplying actual values for any parameters expected, according to the definition of the input message in the BPEL program. Meanwhile, the engine offers an administration interface which supports browser-based-monitoring of existing workflow executions.

The translator component implemented for BPEL thus edits not just the BPEL code but also the appropriate WSDL definitions so the engine "thinks it is correct to include the added parameters, workflowID etc".

B. Configuration

For the purpose of the experiments described here, two simple workflows are defined. The first, *OneCall* encapsulates only a single service invocation. On receipt of a request, *OneCall* invokes a single service **Calc** before replying to the requester with the response from **Calc**. A user parameter which appears as an element *value* in the XML request is passed to the called service. In a second example *TwoCallAnd*, the invocation of **Calc** is only made if the response from a prior call to a separate service **Test** returns *true*. The return result of **Calc** is a *float* value and *0.0* is a suitable default to return in the event **Calc** is not invoked. The two example workflows are shown diagrammatically in figure 4



(a) *OneCall* (b) *TwoCallAnd*

Fig. 4. Illustrations of workflows used in the experiments.

For the purposes of the experiments, implementations of *Test* and *Calc* are defined as configurable, CPU-heavy loads. Each has a single operation, *test* and *calc* respectively, which is invoked in these experiments. In both cases, a dummy calculation is run in a loop for a configurable number of iterations. While *calc* simply returns the result of the dummy calculation, *test* returns a boolean value, with the proportion of *true* returns being configurable.

C. Experiment plan

In these initial experiments a range of single workload scenarios are implemented. The implementations of *Test* and *Calc* are employed by the workflows *OneCall* or *TwoCallAnd* in individual workloads of 100 requests. During such a workload, a change, e.g. in request rate, may be initiated, which prompts the adaptivity support to react in such a way as to demonstrate some adaptation in a controlled way. Early examples demonstrate manipulation of subpool size in response to changing workload and subsequent examples demonstrate changes in invocation mapping in response to changes which arise in the ranking of alternate services.

To facilitate measurement of the results, the experiments are contrived so that in all cases the significant effect can be observed in terms of response time, either the total time to execute the whole workload, or in some cases the time to execute individual requests within a particular workload. The effect is achieved in some later experiments by arranging for the response of an individual service invocation to be proportional to a non-functional property under consideration.

The prototype adaptive workflow engine described in section IV is deployed on a private workstation where a collection of shell scripts and utilities control the generation of test load and collection of results in a database. Specifically, the overall response times for a total workload, and for each individual request within each workload are noted. The adaptive engine manages a collection of machines within a shared cluster, which is located at the same institution. The private and shared workstations have respectively 3GHz Pentium 4 processor with 1 GB memory and 2.8 GHz Xeon processors with 2 GB main memory. The individual experiments are describe in subsequent subsections.

D. Executing a constant rate load

By way of calibration, some initial measurements of the handling of constant rate loads are presented. Such loads are important because they represent the ideal for a service provisioning scenario. The simpler of the two workflows *OneCall* is employed in this scenario. In these experiments, the duration of a single invocation of *calc* has a duration of 48 seconds on one of the cluster machines.

Figure 5 show how the response to constant-rate workloads varies as the number of machines made available to the user is varied. The results are shown equivalently through total elapsed time for completion of the whole workload and average response time per individual request. The former shows clearly the speedup (about 2.4) achieved through multi-processing within a single Xeon machine, but also that a significant speedup is achieved overall in what is not a highly tuned setup. The latter graph shows how submitting requests at a higher rate than allowed for by the configured pool size can have a very significant effect on the response time of individual requests. The extreme example is where the average response time for requests submitted to a single machine at a rate of 12/48 requests per second is over 900 seconds.

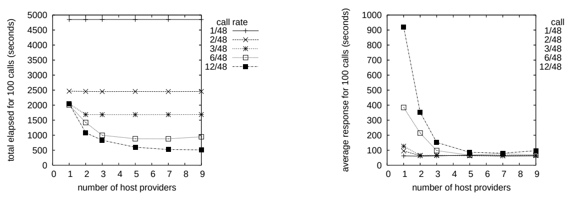


Fig. 5. Measured performance of prototype workflow engine under constant-rate loads, each load comprising 100 requests of *OneCall*.

E. Varying user request rate

In this experiment, the input rate is kept constant for a time, but increased suddenly at a certain point after the start of the workload and then kept constant again for the remainder of the workload. The experiment is intended to demonstrate feed back based control of subpool size. Receiving updates from the probe giving recent measurements of the workflow request and response rates, three simple policies are implemented for comparison

maximize forces pool size immediately to the total number of machines available for the workload. This strategy effectively over-provisions so as to meet the per-request response time requirement, but at the cost of wasting resources.

follower implements a simple algorithm shown in figure 6 with a view to making short term output rate follow short term input rate. The algorithm seeks to make a change in pool size when the difference between input and output rates suggests that such a change might be worthwhile, but also makes allowance for startup and shutdown conditions, where output or input rate is zero.

disable disables changes to the pool size. In this case, the pool size remains at its minimum value of 1. This strategy reflects the situation where the provider provisions just enough resources to meet the nominal average request rate. The strategy avoids wastage of resources through over-provisioning, but may cause the user to experience degraded performance at times, due to short term variations in request rate.

```
sreq = inputRate.shortTerm();
sresp = outputRate.shortTerm();
lreq = inputRate.longTerm();
lresp = outputRate.longTerm();

if (sreq > 0) {
    if (sresp/sreq < ((double)poolsz)/(poolsz+1)) {
        if (lresp > 0.0) poolsz += 1;
    } else if (sresp/sreq > ((double)poolsz)/(poolsz−1)) {
        poolsz −= 1;
        if (poolsz < 1)
            poolsz = 1;
    }
} else if (lresp()/lreq < ((double)poolsz)/(poolsz+1))
        poolsz += 1;
}
```

Fig. 6. Example feedback strategy *follower*.

Figure 7 shows how the response to the example step change in the request rate varies as the point at which that step change occurs is changed though the duration of the workload, for the different control strategies. The rate of 1/48 requests is equivalent to a serial execution, so even in a single multi-CPU machine, only a single CPU will be busy. The higher rate of 6/48 is sufficient to occupy 6 CPUs. Since the total amount of work to be done is constant, the use of parallelism, including that attained by increasing the pool size, permits faster completion of the

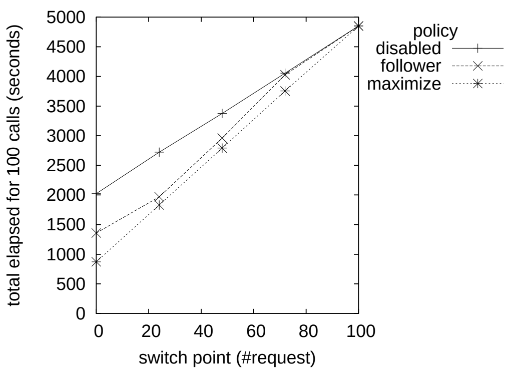


Fig. 7. Measured performance of adaptive workflow engine under load which changes from constant rate of 1/48 to 6/48 requests per second after request *switch point*, each load comprising 100 requests of *OneCall*.

overall workload. Thus, the earlier the rate switch occurs, the greater the potential for parallel speedup and so the faster is the overall completion.

Figure 8 shows the impact which the adaptive control can have on individual runs, clearly offering a significant improvement.

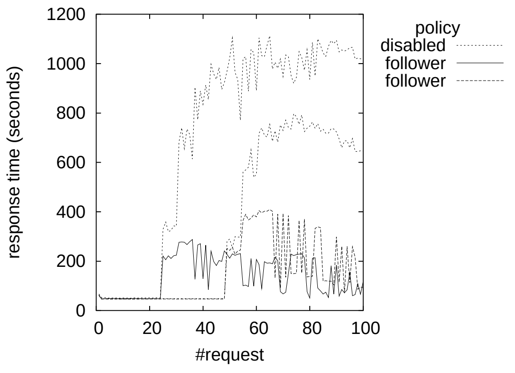


Fig. 8. Measured responses for individual requests within example workloads where the request rate changes from constant rate of 1/48 to 6/48 after request number 24 and 48.

These responses show what benefit might be achieved by the wasteful *maximize* strategy, and that the simple *follower* strategy gains much of that benefit without the wastage. Clearly the benefit is greater when the switch between request rates is made early during the workload, but as shown in Figure 8, an improvement by a factor of 2 in the per-request response time can be achieved when the rate switch occurs half way through the workload. Clearly, the potential benefit of the adaptive strategy is in permitting better handling of more variable workloads.

F. Varying induced load for constant user request rate

In the case of any complex application, it is to be expected that the response time may vary dramatically from one request to the next. This effect is manifested in workflow programs where a small change in the value of a variable used in a workflow level control structure can alter the set of invocations made. In this experiment, the workflow request rate is kept constant, but the number of component service invocations made by those requests is varied. This in turn varies the load induced by those requests, though the input request rate is constant. In this scenario, $TwoCallAnd$ is employed, with the cost of the invocation of *calc* being 10 times the cost of the invocation of *test*. The rate induced load is varied by varying the frequency of *true* results from the cheaper *test* invocation.

The next experiment uses $TwoCallAnd$ to show the response to a dynamic change in this induced load during a workload. To this end, control of the result pattern of *test* is effected by means of an extra parameter to the call, so that different patterns can be selected for different runs. In fact, a similar approach is adopted towards setting other parameters governing component service execution. Figure 12 shows the response to a step change in the load

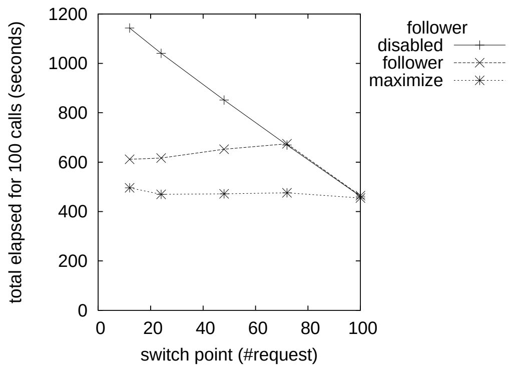


Fig. 9. Measured performance of adaptive workflow engine under constant rate of 6/24 requests of *TwoCallAnd* per second where the induced load changes at *switch point* as the proportion of *true* results returned by *Test* changes from 0.08 to 0.5

induced by the two-invocation workflow under the three alternative feedback strategies. In this case, the overall response has a different shape from that seen in figure 8 because the switch in workflow behaviour affects the total load imposed on the available resources. In the case of *maximize* where spare parallel resources are available all the time the extra load imposed is soaked up without affecting the overall completion time. In the case where no parallelism is permitted, the performance is worse the earlier the switch takes place during the workload. However, as in the case of the step change in request rate, employing the *follower* strategy achieves much of the improvement which is achievable, as indicated by the *maximize* strategy. Figure 12 shows the variation of individual runs within two example cases. As before, a significant gain is made in per-request response times through the use of the feedback mechanism.

G. Adding an alternative service

Remaining experiments demonstrate manipulation of the invocation mapping in response to changes relating to

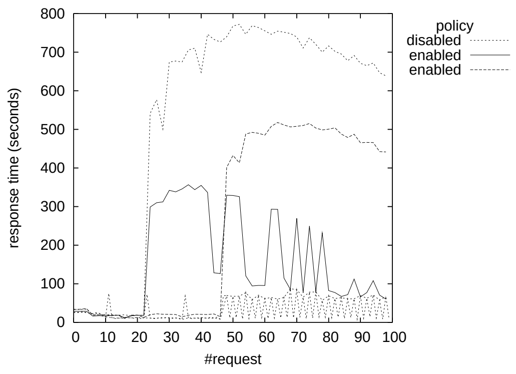


Fig. 10. Measured performance of adaptive workflow engine under constant rate of 6/24 requests of *TwoCallAnd* per second where the induced load changes at *switch point* as the proportion of *true* results returned by *Test* changes from 0.08 to 0.5. The graph shows the variation of individual request response time for *switch point* values of 24 and 48, plotted for every *other* request. Only alternate points are plotted because of the high variability which arises even after the switch point where only half of the requests are invoking the *calc* operation. This variability is shown additionally in the case of the *follower* strategy for *switch point* value of 48.

non-functional properties. For the purposes of these experiments two alternative implementations of the *Calc* service are defined with the properties shown in Table I.

	monetary cost	result precision	runtime
Calc000	4.5	10	× 1.0
Calc001	0.5	1	× 0.1

TABLE I
Alternative services for use in QoS experiments.

In addition to the cost and precision parameters, the runtimes are seen to be roughly in proportion to the non-functional properties. Essentially, the alternatives are a monetarily and computationally expensive version that offers better result precision and a cheaper alternative that offers lower result precision. For these initial results, having the computational cost mirror non-functional properties is convenient since a change in the non-functional property will be portrayed in the response time domain.

The first experiment applies a constant rate workload of *TwoCallAnd* requests, in the presence of just the most expensive implementation of *Calc*, *Calc000*, but installs the cheaper alternative *Calc001* during the execution while the mapping policy is to always select the monetarily cheapest service. The measured results are shown in figure 12. In this experiment, the subpool size is fixed at 2 machines, and the frequency of *true* response from *Test* is fixed at 50%. The results show that the earlier introduction of the cheaper alternative does tend to correspond to a reduction in the overall response time. However, there is obvious levelling off of response if the cheaper service is made available very early. When the request rate is. 6/24, the time to submit all 100 requests is 100*(24/6) = 400 seconds. Clearly

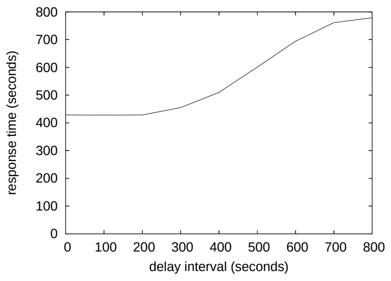


Fig. 11. Measured behaviour of adaptive workflow engine under constant rate of 6/24 requests of *TwoCallAnd* per second where an alternative (cheaper) implementation of **Calc** is registered at some interval (*delay interval*) after the start of the workload.

this is a lower bound on the achievable response time.

H. Choosing between alternate service implementations

The final experiment seeks to demonstrate a more complex policy, which combines data from the two probes. In this experiment, the policy chooses dynamically whether to maximize result precision or to minimize cost, depending on the average cost per request so far during the workload. In the scenario suggested by this experiment the requirement is to maximize result precision, while incurring no more than a specified average monetary cost. The need to adapt arises because the actual pattern of execution of the *TwoCallAnd* workflow includes more invocations of the expensive *Calc* service than can be afforded. In the experiment, the frequency of *true* returns from *test* is initially low enough that the cost bound is not exceeded by using the more expensive *Calc000*. However, at some point after the start of the workload, the frequency of *true* returns from *test* is increased so that use of *Calc000* is no longer affordable. The scenario is contrived to require a metric which combines the separately generated statistics of workflow requests and service invocations. Specifically, to compute the average cost so far of requests on a particular workflow, it is desirable to obtain the count of completed workflow requests and the count of completed service invocations, within those complete workflow requests. The adaptivity component can receive incrementally from the two probes all event data, and thereby construct the counts it needs. However, in this experiment an alternative strategy is employed. The adaptivity component uses instead the already aggregated counts of workflow requests, and of invocations of particular services. Figure 12 shows the measured results. Recalling that the duration of an invocation to either service implementation is contrived to be proportional to the corresponding result precision value, and nearly proportional to the cost per invocation, it is clear that the *upperbound* policy is controlling the average cost yet also including some of the more expensive calls with higher precision.

Recalling that the response time for the basic version of *calc* is 48 seconds, that for the cheaper version is 4.8 seconds. The maximum ratio of *true* returns from *test* is 0.5, where there are 50 calls to *calc*. Where *min cost* policy

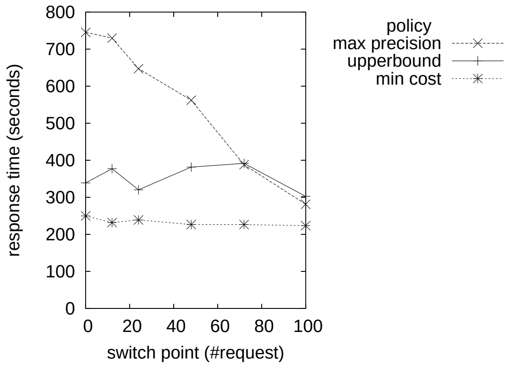


Fig. 12. Measured behaviour of adaptive workflow engine under constant rate of 6/12 requests of *TwoCallAnd* per second and three different policies. There are two alternative implementations of **Calc**, with different monetary cost and result precision metrics. At some point after the start of the workload, i.e. with a certain request number, the proportion of true results returned by *test* is increased from 0.08 to 0.5. The three policies are: (1) always maximize precision; (2) always minimize cost; and (3) if average cost so far is less than 0.5 per workflow request, maximize precision, else minimize cost.

is used, the time cost of for these 50 calls is $50 \times 4.8 = 240$ seconds; 120 seconds with parallelism of 2. In this case, the performance is limited by the request rate, as described in Section G. Under the *upperboundcost* policy, maximum response time is 400 seconds. If n_0 and n_1 are the number of calls to *Calc000* and *Calc001* respectively, $48n_0 + 4.8n_1 = 400$. Since the graph is coincident with that of the *max precision* case, the ratio of *true* returns from *test* at this point is 0.5. Then $n_0 + n_1 = 50$. Solving these two equations, gives $n_0 = 6$ and $n_1 = 44$. Recalling that the monetary costs of n_0,n_1 are 4.5 and 0.5 respectively, the total monetary cost in this case is $6 * 4.5 + 44 * 0.5 = 49$, so the average cost over the 100 workflows is 0.49 per request, which fits ok with the bound 0.5.

It seems in this scenario, the adoption of an approximation has turned out to be reasonable. One aim of future work will be to investigate further the trade-off between the accuracy of statistics and the costs associated with their creation.

VI. Conclusions

This paper has outlined a way to construct an adaptive workflow engine by combining a non-adaptive workflow engine and a dynamic service deployment facility, together with a small amount of code to implement monitoring and control functions. The result builds on the strengths of the third-party components, re-using the workflow engine unchanged and only requiring a limited interface to the deployment service. One advantage of the approach is the reduced development cost. Another is that the support for adaptation is cleanly separated from the re-used components. It is hoped that this partitioning will make it easier to investigate more complex strategies for adaptation, which involve multiple mechanisms that may interact. Another benefit is the potential for re-using the wrapper components around an alternative basic workflow engine.

Of course the narrow interface between adaptivity support and basic components may be a limiting factor. However, experiments so far with the prototype implementation have demonstrated a number of scenarios where the adaptivity mechanisms have proved successful. Specifically, the prototype engine has shown itself able to make beneficial adaptations in both resource pool and service invocation mapping. The prototype appears also to have pointed to an interesting trade-off in the selection and use of the statistics which underlie adaptive policies.

References

[1] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *SCC*. IEEE Computer Society, 2004.

[2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano-sla based management of a computing utility. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 855–868, 2001.

[3] Luciano Baresi, Elisabetta Di Nitto, Carlo Ghezzi, and Sam Guinea. A framework for the deployment of adaptable web service compositions. *SOCA*, 1:75–91, 2007.

[4] Alberto Bartoli, Ricardo Jimnez-Peris, Bettina Kemme, Cesare Pautasso, Simon Patarin, Stuart Wheater, and Simon Woodman. The adapt framework for adaptable and composable web services. *Distributed Systems On Line*, 2005.

[5] Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *ICAPS*, volume 3165 of *Lecture Notes in Computer Science*, pages 153–163. Springer, 2004.

[6] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Syst. J.*, 43(1):136–158, 2004.

[7] Catalin Dumitrescu, Ioan Raicu, and Ian Foster. Di-gruber: A distributed approach to grid resource brokering. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Active Endpoints. Activebpel engine. http://www.activebpel.org/, 2007.

[9] Abdelkarim Erradi and Piyush Maheshwari. Adaptivebpel: a policy-driven middleware for flexible web services composition. In *MWS*, 2005.

[10] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Jr, and Hong-Linh Truong. Askalon: a tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, 17:143–169, 2005.

[11] Grimoires. Grid registry with metadata oriented interface: Robustness, efficiency, security. http://twiki.grimoires.org/bin/view/Grimoires/, 2005.

[12] Richard Hull and Jianwen Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2):86–95, 2005.

[13] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending bpel for run time adaptability. In *EDOC*. IEEE Computer Society, 2005.

[14] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] Avraham Leff, James T. Rayfield, and Daniel M. Dias. Service level agreements and commercial grids. *Internet Computing*, July–August 2003.

[16] OASIS. Uddi version 3.0.2. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm, 2004.

[17] OASIS. Web services business process execution language version 2.0, April 2007.

[18] J. Palmer and I. Mitrani. Optimal and heuristic policies for dynamic server allocation. *Journal of Parallel and Distributed Computing*, 65:1204–1211, 2005.

[19] S.R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *WWW*. IEEE Computer Society, 2002.

[20] Kavitha Ranganathan and Asit Dan. Proactive management of service instance pools for meeting service level agreements. In *ICSOC*, pages 296–309, 2005.

[21] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad P. A. van Moorsel, and Fabio Casati. Automated sla monitoring for web services. In *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 28–41, London, UK, 2002. Springer-Verlag.

[22] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 331–340, New York, NY, USA, 2007. ACM Press.

[23] Paul Watson, Chris Fowler, Charles Kubicek, Arijit Mukherjee, John Colquhoun, Mark Hewitt, and Savas Parastatidis. Dynamically deploying web services on a grid using dynasoar. In *ISORC*, pages 151–158, Gyeongju, Korea, April 2006. IEEE Computer Society.

[24] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. Qos-based scheduling of workflow applications on service grids. In *e-Science*. IEEE Computer Society, 2005.

[25] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and henry Chang. Qos-aware middleware for web service composition. *Transactions on Software Engineering*, 30(5):311–327, may 2004.