myjournal manuscript No.
(will be inserted by the editor)

Software Architecture Definition for On-demand Cloud Provisioning

Clovis Chapman · Wolfgang Emmerich · Fermín Galán Márquez · Stuart Clayman · Alex Galis

Received: date / Accepted: date

Abstract Cloud computing is a promising paradigm for the provisioning of IT services. Cloud computing infrastructures, such as those offered by the RESERVOIR project, aim to facilitate the deployment, management and execution of services across multiple physical locations in a seamless manner. In order for service providers to meet their quality of service objectives, it is important to examine how software architectures can be described to take full advantage of the capabilities introduced by such platforms. When dealing with software systems involving numerous loosely coupled components, architectural constraints need to be made explicit to ensure continuous operation when allocating and migrating services from one host in the Cloud to another. In addition, the need for optimising resources and minimising over-provisioning requires service providers to control the dynamic adjustment of capacity throughout the entire service lifecycle. We discuss the implications for software architecture definitions of distributed applications that are to be deployed on Clouds. In particular, we identify novel primitives to support service elasticity, co-location and other requirements, propose language abstractions for these primitives and define their behavioural semantics precisely by establishing constraints on the relationship between architecture definitions and Cloud management infrastructures using a model denotational approach in order to derive appropriate service management cycles. Using these primitives and semantic definition as a basis, we define a service management framework implementation that supports on demand cloud provisioning and present a novel monitoring framework that meets the demands of Cloud based applications.

Keywords Cloud Computing, Service Definition, Software Architecture, Service Management, Monitoring System

This research has been partly funded by the RESERVOIR EU FP7 Project through Grant Number 215605.

Clovis Chapman · Wolfgang Emmerich
Dept. of Computer Science, UCL, Gower Street, London, UK
E-mail: c.chapman@cs.ucl.ac.uk E-mail: we@acm.org

Fermín Galán Márquez
Telefónica I+D, Emilio Vargas 6, Madrid, Spain
E-mail: fermin@tid.es

Stuart Clayman · Alex Galis
Dept. of Electronic Engineering, UCL, Gower Street, London, UK
E-mail: s.clayman@ee.ucl.ac.uk E-mail: a.galis@ee.ucl.ac.uk

1 Introduction

Cloud computing [35] is a promising paradigm for the provisioning of IT services. Cloud computing infrastructures, such as those offered by the RESERVOIR project [25], aim to facilitate the deployment, management and execution of services across multiple physical locations in a seamless manner.

Until recently, operating systems managed the allocation of physical resources, such as CPU time, main memory, disk space and network bandwidth to applications. Virtualisation infrastructures, such as Xen [4] and VMWare [31] are changing this by introducing a layer of abstraction known as a *hypervisor*. A hypervisor runs on top of physical hardware, allocating resources to isolated execution environments known as *virtual machines*, which run their own individual virtualised operating system. Hypervisors manage the execution of these operating systems, booting, suspending or shutting down systems as required. Some hypervisors even support replication and migration of virtual machines without stopping the virtualised operating system.

It turns out that the separation between resource provision and operating systems introduced by virtualisation technologies is a key enabler for Cloud computing. Specifically, virtualisation is an enabler for Infrastructure-as-a-Service (IaaS) Clouds, the type on which this paper is fo-

cused. For a description of the different types of Cloud computing see [35].

Compute Clouds provide the ability to lease computational resources at short notice, on either a subscription or pay-per-use model and without the need for any capital expenditure into hardware. A further advantage is that the unit cost of operating a server in a large server farm is lower than in small data centres. Examples of compute Clouds are Amazon's Elastic Compute Cloud (EC2) [1] or IBM's Blue Cloud [15]. Organisations wishing to use computational resources provided by these Clouds supply virtual machine images that are then executed by the hypervisors running in the Cloud, which allocate physical resources to virtualised operating systems and control their execution.

With an increasing number of providers seeking to migrate services to the Cloud in order to save on deployment costs, cater for rapid growth or generally relieve themselves from the responsibility of provisioning the infrastructural resources needed to support the service, whether related to power, bandwidth, software or hardware [3], there is a crucial need to ensure that a same service quality can be retained when relying upon Clouds while generally delivering on the promise of lowering costs by minimising overprovisioning through efficient upscaling and downscaling of services.

In this paper we review the implications of the emergence of virtualisation and compute Clouds for software engineers in general and software architects in particular. We find that software architectures need to be described differently if they are to be deployed into a Cloud. The reason is that scalability, availability, reliability, ease of deployment and total cost of ownership are quality attributes that need to be achieved by a software architecture and these are critically dependent upon how hardware resources are provided. Virtualisation in general and compute Clouds in particular provide a wealth of new primitives that software architects can exploit to improve the way their architectures deliver these quality attributes.

The principal contribution of this paper is a discussion of architecture definition for distributed applications that are to be deployed on compute Clouds. The key insight is that the architecture description needs to be reified at run-time so that it can be used by the Cloud computing infrastructure in order to implement, monitor and preserve architectural quality goals. This requires several advances over the state of the art in software architecture. Architectural constraints need to be made explicit so that the Cloud infrastructure can obey these when it allocates, replicates, migrates and deactivates virtual machines that host components of the architecture. The architecture also needs to describe how and when it responds to load variations and faults; we propose the concept of elasticity rules for architecture definitions so that the Cloud computing infrastructure can replicate components and provide additional resources as demand grows or components become unavailable. Finally, there must be an understanding of how a management cycle for a service deployed on a Cloud can be derived from a description of these constraints and we demonstrate how this can be achieved using a *model driven approach*.

As such the overall contributions of the paper are as follows: we identify a list of requirements and constraints that a provider must describe when deploying and hosting a multi-component application on a Cloud. We present a language for the description of such requirements that builds on existing standards and introduces new abstractions such as the ability to specify on demand scaling. We define an architecture for a Cloud infrastructure to support these abstractions and specify clear behavioural semantics for our language with respect to this architecture using a model denotational approach. We then describe the overall implementation of the service lifecycle management process, with particular focus on service management components and the monitoring infrastructure which will address the scaling requirements and physical distribution of application services. Finally we evaluate our language primitives experimentally with a distributed computational chemistry application.

This paper is an enhancement and extension to work previously presented [5] and is structured as follows: In Section 2, we present the background to this research and in particular the primitives that are now provided by modern virtualisation and Cloud computing technologies in general and the infrastructure developed by the RESERVOIR project in particular. We then use a motivating example in Section 3 to argue why architecture definition for Cloud computing differs from more conventional deployments. In Section 4, we describe our novel abstractions, such as co-location constraints and elasticity rules for describing architectures that are to be deployed in a Cloud. Section 5 presents the main components of the architecture which manage the lifecycle of services within a cloud, namely the Service Manager and the Monitoring Framework. In Section 6 we describe the experimental evaluation of our approach by means of a computational chemistry application that we have deployed in a computational Cloud. We discuss related work and present conclusions in Sections 7 and 8.

2 Background

The *Resources and Services Virtualization without Barriers* (RESERVOIR) project is a European Seventh Framework Programme (FP7) project which aims to promote through standardisation an open architecture for federated Cloud computing. It defines an open framework of modular components and APIs that enable a wide range of configurations within the Cloud space, focusing on Infrastructure as a Service (IaaS) Clouds [35].

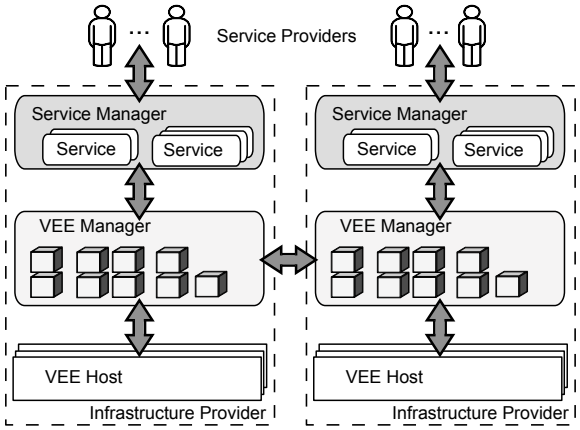

Fig. 1 RESERVOIR Architecture

The RESERVOIR architecture [25] aims to satisfy the vision of service oriented computing by distinguishing and addressing the needs of *Service Providers*, who understand the operation of particular businesses and offer suitable *Service* applications, and *Infrastructure Providers*, who lease computational resources in the form of a Cloud computing infrastructure. This infrastructure provides the mechanisms to deploy and manage self contained services, consisting of a set of software components, on behalf of a service provider.

The Cloud computing abstractions supported by this infrastructure and the architecture we have used to implement these abstractions are described in detail in [26]. The architecture is shown in Figure 1.

RESERVOIR builds upon generally available virtualisation products and hypervisors. The lowest layer of the RESERVOIR architecture is the *Virtual Execution Environment Host* (VEEH). It provides plugins for different hypervisors and enables the upper layers of the architecture to interact with heterogeneous virtualisation products. The layer above is the *Virtual Execution Environment Manager* (VEEM), which implements the key abstractions needed for Cloud computing. A VEEM controls the activation of virtualised operating systems, migration, replication and deactivation. A VEEM typically controls multiple VEEHs within one site. The key differentiator from other Cloud computing infrastructure is RESERVOIR's ability to federate across different sites, which might be implementing different virtualisation products. This is achieved by cross-site interactions between multiple different VEEMs operating on behalf of different Cloud computing providers. This supports replication of virtual machines to other locations for example for business continuity purposes. The highest level of abstraction in the RESERVOIR architecture is the *Service Manager*. While the VEEM allocates services according to a given placement policy, it is the Service Manager that interfaces with the Service Provider and ensures that requirements (e.g. resource allocation requested) are correctly enforced. The Service Manager also performs other service management tasks, such as accounting and billing of service usage.

An implementation of the VEEM layer, OpenNebula, which we have used in our experimental evaluation described below is publicly available from [24]. It provides the ability to interact with hypervisors such as Xen [4] or VMWare [31] via appropriate plugins and is included from release 9.04 in the Ubuntu Linux distribution.

3 Motivation

How is a software development project now going to use Cloud computing infrastructures, such as the one provided by RESERVOIR. We aim to answer this question and then derive the main research questions for this paper using a running example, an enterprise resource planning system, such as those provided by SAP [27]. A high-level architecture of an SAP system is illustrated in Figure 2. SAP ERP systems have a multi-tiered software architecture with a relational database layer. On top of the database is an application layer that has a *Central Instance*, which provides a number of centralised services, such as synchronisation, registration and spooling capabilities, and generally serves as a database gateway. Moreover SAP applications have a number of *Dialog Instances*, which are application servers responsible for handling business logic and generating HTTP-based dialogues that are shown in a browser. A *Web Dispatcher* may be used to balance workloads between multiple dialog instances. To date SAP systems are hosted in substantial data centres, but organisations (playing the role of Service Provider) in the future might wish to deploy it in a compute Cloud in order to avoid the significant capital investment associated with the construction of a data centre.

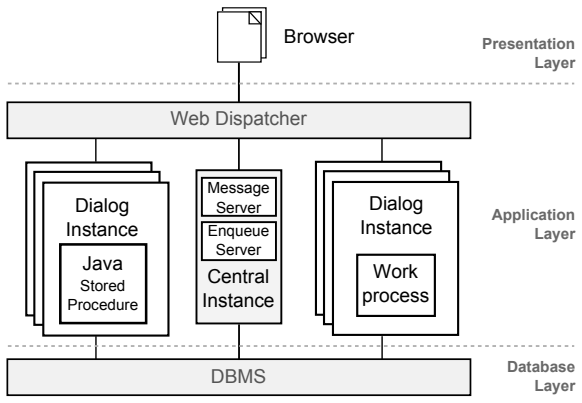

Fig. 2 SAP Three-Tiered Architecture

If the SAP system is handed over to a Cloud computing provider that uses an infrastructure based on the RESER-

VOIR architecture, a number of architectural constraints of the SAP system will need to be obeyed by the Cloud. For example, the Central Instance will frequently make queries to the database and in a typical SAP configuration the Central Instance and the database need to be co-located on the same LAN subnet. When the VEEM allocates the virtual machines for the Central Instance and the DBMS to particular VEEHs, it will have to respect this co-location constraint. Another architectural constraint is that the Central Instance can not be replicated in any SAP system. Dialog Instances, on the other hand are replicated to accommodate growing demand. Therefore these architectural constraints have to be expressed by the SAP provider and made available to the Service Manager so that it can obey these constraints at runtime.

This requires the software architecture to be expressed in terms of services, their relationship and their resource requirements. While a traditional SAP deployment would require that spare capacity is retained to deal with peaks in demand, Cloud computing introduces the ability to minimise overprovisioning by tying resource provision directly to the needs of the application. Resource requirements, in this case dictated by the number of Dialog Instances required to handle the current load, can be scaled dynamically and adjusted to maximise cost savings. This requires means of describing the state of the system and rules for adjustments.

4 Architecture definition

4.1 Requirements

Based on an understanding of the underlying hosting infrastructure described in Section 2, and the example provided in Section 3, we can break down the core issues that must be defined when deploying and running software systems on a Cloud computing infrastructure into the following high-level requirements, in order to provide a suitable definition for the terminology employed in later sections:

MDL1 Software composition: *A software system may be composed of one or more* loosely coupled components, *which may have differing* resource *(e.g. CPU, memory) and* software *(e.g. operating system, libraries, disk image) requirements*.

The components of the multi-layered SAP system, the Web Dispatcher, Central Instance, Dialog Instance and DBMS, will have varying hardware and software requirements, but will nevertheless be required to be managed jointly. We can expect for example the DBMS service to be very I/O and memory intensive and with large storage requirements. In contrast, the Dialog Instances may be more processor intensive, and hardware requirements may be adjusted accordingly.

MDL2 Network topology: *The system may require a specific* network topology *both to interconnect components of the system and communicate with external systems.*

With respect to the SAP system, the Web Dispatcher should provide an external interface and internal components should be at the very least interconnected, though external access may not necessarily be required.

MDL3 Capacity adjustment: *Hardware requirements may evolve during the lifetime of the system according to workload, time or other application-level variables.*

In order to deal with potential increases in requests, it may be necessary to deploy additional Dialog Instances in order to facilitate load balancing and ensure a certain level of performance.

MDL4 Dependencies: *Deployment and un-deployment dependencies may exist between components.*

The order in which components of an SAP system are started or stopped may affect the overall operation of the system. The DBMS and Central Instance components, serving as the backbone of the system, should be active before individual Dialog Instances.

MDL5 Location constraints: *Constraints on the distribution of service components across physical locations may exist.*

Federation of Clouds is key to enabling scalable provisioning of services. However along with the ability to seamlessly deploy services across multiple physical and administrative domains comes a need to allow service providers to control the “spread” of the application by defining clear constraints on the distribution of services across sites. These constraints can be of a technical nature (e.g. deploy certain components on a same host) or administrative (e.g. avoid un-trusted locations). Though we have, for example, established that the Central Instance and DBMS should be located on a same (virtual) network, a service provider may wish to minimise latency by ensuring proximity.

MDL6 Customisation: *Components may be dependent on configuration parameters not known until deployment.*

When deploying multiple instances of a same component, certain application-level parameters may be instance specific. As such it may be necessary to customise individual instances upon their creation and deployment. Dialog Instances may for example require the IP addresses of the Central Instance and DBMS to be provided, if this information is not know at pre-deployment time (e.g. dynamic IP allocation via DHCP).

In order to automate the management of a software system on a Cloud infrastructure it is necessary for a service provider to communicate both the software system stack

(OS, middleware, application, configuration, and data) providing self contained services in the form of a virtualised image (addressing requirement MDL1) and a description of these requirements in the form of a *Service Definition Manifest* (addressing requirements MDL2-MDL6). The manifest therefore serves as a contract between service and infrastructure providers regarding the correct provisioning of a service. It hence reifies key architectural constraints and invariants at run-time so that they can be used by the Cloud.

To define manifests, we require a declarative language whose syntax should be sufficiently flexible to cater for a general purpose service provisioning environment, and provide the necessary abstractions to describe capacity and operational requirements of the software architecture both at deployment time and throughout the entire lifecycle.

We rely in our implementation on the *Open Virtualisation Format* (OVF) [10], a DMTF standard backed by VMWare and XenSource which aims to offer a packaging mechanism in a portable and platform neutral way. Building on open standards facilitates interoperability particularly in the context of federation and eases compatibility with existing services and tools. In addition it ensures that as Cloud technology matures, continued compliance with the standard avoids vendor lock-in and potential deployment on newer platforms. OVF hence serves as a building block for our manifest, and provides the syntax for the description of virtual disks, networks, resource requirements and other issues related to dependencies or customisation. However, OVF (as other service description languages for existing virtualisation technologies) primarily caters for the initial distribution and deployment of fixed size services [10], which does not by itself fully realise the vision of Cloud computing.

Indeed, Clouds differ from traditional software deployment in many ways. Beyond the impact of virtualisation on multi-component architectures, existing deployment mechanisms are typically one-way "channels" where a service is configured and deployed according to an initial deployment descriptor. There is no feedback mechanism to communicate specific state, parameters and other information from a deployed service back to the infrastructure to adapt the execution environment dynamically. The manifest should enable the automation of provisioning and management through template based provisioning, where the service manifest is used as a template for easily provisioning instances of the application, and support for resource consumption control.

We hence need to add a number of abstractions to OVF, the primary being **elasticity specification** in the form of rules allowing conditions related to the state and operation of the service, such as application level workload, and associated actions to follow should these conditions be met, **application domain description**, which allow the state of the application to be described in the form of monitorable application level parameters and **placement and co-location constraints**, which identify sites that should be favoured or avoided when selecting a location for a service.
In previous work [13], we have discussed a number of additional extensions to the OVF syntax to support Clouds, including attribute and section changes to incorporate support for service components IDs in elastics arrays, cross virtual machines reference, IP dynamic addresses and elasticity rules and bounds. However, a syntactic definition of a deployment descriptor only forms part of what is necessary to ensure these requirements are met with respect to the underlying Cloud computing infrastructure.

Indeed, there must exist a clear understanding of how we derive from the language used to express the requirements of the Service Provider a *management cycle*, which will consist of several actions being taken throughout the lifetime of a service to ensure a certain service quality being obtained. Using the RESERVOIR framework as a reference, and examining specifically issues related to dynamic capacity adjustment and service deployment, we now describe how the *behavioural semantics* for our manifest language are described and how they guide the operation of underlying Cloud components.

Focusing specifically on elasticity and application domain description, as well as service deployment, we refine and extend in this paper our OVF based service definition language syntax to incorporate these abstractions.

4.2 Manifest Language Definition

In this section, we describe the overall approach undertaken to define and provide support for the *Manifest Language*. This is achieved through the specification of three complementary facets of the language: the abstract syntax, the well-formedness rules, and the behavioural semantics. The abstract syntax of the manifest language is modelled using the *Essential Meta-Object Facility* (EMOF), an OMG standard part of the Model Driven Architecture initiative [22] for describing the structure of meta-data, and embedded within an object-oriented model of the RESERVOIR architecture. Because the manifest describes the way in which a RESERVOIR based infrastructure should provision a service application, the semantics of the language can be expressed in the model denotational style to define semantics that we introduced in [29] as constraints between the abstract syntax and domain elements that model the operation of Cloud infrastructure components. These constraints are formally defined using the *Object Constraint Language* (OCL) [23], a language for describing consistency properties, providing the static and behavioural semantics of the language. In this manner the language serves to constrain the behaviour of the underlying infrastructure, ensuring the correct provisioning of the software system services.

The motivations for this approach are two-fold: firstly by modelling the syntax of the manifest language as an EMOF model, we seek to express the language in a way that is independent of any specific implementation platform. Components of a Cloud infrastructure such as RESERVOIR may rely on a number of different concrete languages, whether implementation languages (Java, C++, etc.), higher-level "meta" languages (HUTN, XML, etc.), or even differing standards (WS-Agreement, OVF, etc.). A higher level of abstraction ensures that we free ourselves from implementation specific concerns, and allows seamless and automated transitions between platform specific models as required by components.

Secondly, providing a clear semantic definition of the manifest using OCL allows us to identify functional characteristics that service management components should present in order to support capabilities such as application based service elasticity, again irrespective of the implementation platform. As such the definitions presented in this paper extend beyond the scope of RESERVOIR or any specific Cloud infrastructure, instead providing a clear understanding of expected provisioning behaviours, with respect to identified and required component interfaces.

Finally, we may also consider that clear semantics ensure that we limit ambiguities when it comes to interpretation of the manifest. This is of crucial importance where financial liabilities may exist; a formal understanding of the nature of the service being provided is required in order to ensure that the service is provisioned as expected by both parties, and in a way that both can evaluate to be correct, either through run-time monitoring capabilities or historical logs. We achieve this by tying the specification of the manifest to the underlying model of the Cloud infrastructure.

4.2.1 Abstract syntax

The abstract syntax of the manifest describes the core elements of the language and their accompanying attributes. The core syntax relies upon, as previously stated, OVF [10]. The OVF descriptor is an XML-based document composed of three main parts: description of the files included in the overall service (disks, ISO images, etc.), meta-data for all virtual machines included, and a description of the different virtual machine systems. The description is structured into various "Sections". Focusing primarily on the most relevant, the `<DiskSection>` describes virtual disks, `<NetworkSection>` provides information regarding logical networks, `<VirtualHardwareSection>` describes hardware resource requirements of service components and `<StartupSection>` defines the virtual machine booting sequence.

Incorporating the OVF standard ensures that we tackle several of the requirements identified in Section 4.1, providing the manifest language with a syntactic model for the expression of physical resource requirements and hardware configuration issues. We introduce new abstractions in the form of extensions to the standard rather than create new independent specifications. OVF is extensible by design and doing so ensures continued compatibility with existing OVF-based systems.

We model these extensions using EMOF. EMOF models are very similar to UML class diagrams, in that they describe classes, the data they contain and their relationships, but are at a higher level of abstraction: they describe the constructs, rules and constraints of a model. As such, EMOF is typically used to define the syntax of languages.

Application description language Reliance on Cloud computing introduces the opportunity to minimise overprovisioning through run-time reconfiguration of a service, effectively limiting resource consumption to only what is currently required by the application. However, when dealing with rapid changes in service context and load, timely adjustments may be necessary to meet service level obligations which cannot be met by human administrators. In such a case, it may be necessary to automate the process of requesting additional resources or releasing existing resources to minimise costs.

This automated scaling of service capacity to support potential variations in load and demand can be implemented in numerous ways. Application providers may implement such scaling at the application level, relying on an exposed interface of the Cloud computing infrastructure to issue specific reconfiguration requests when appropriate. Alternatively, they may have a desire to keep the application design free of infrastructure specific constraints and opt instead to delegate such concerns to the infrastructure itself. With a sufficient level of transparency at the application level for workload conditions to be identified, and through the specification of clear rules associating these conditions with specific actions to undertake, the Cloud computing infrastructure can handle dynamic capacity adjustment on behalf of the service provider.

It is the latter approach that we have chosen to adopt in the context of RESERVOIR. By providing a syntax and framework for the definition and support of elasticity rules, we can ensure the dynamic management of a wide range of services with little to no modification for execution on a Cloud. With respect to the syntax, we can identify the two following subsets of the language that would be required to describe such elasticity: service providers must first be able to describe the application state as a collection of *Key Performance Indicators* (KPIs), and the means via which they are obtained in the manifest. These will serve as a basis for the formulation of the rules themselves, described in the following subsection.

Because we do not want the manifest language to be tied to any specific service architecture or design, it is necessary to decouple the KPI descriptions from the application domain via the use of an *Application Description Language* (ADL). Though it is possible to build elasticity conditions based purely on infrastructure level performance indicators, this may prove limiting. Indeed, the disk space, memory or CPU load may not accurately reflect the current needs of the application, as will be seen in the evaluation. This language will allow the description of services in terms of components, parameters of interest and their monitoring requirements.

Alongside the syntactic requirements, a suitable monitoring framework must exist. A service provider is expected to expose parameters of interest through local *Monitoring Agents*, responsible for gathering suitable application level measurements and communicating these to the service management infrastructure. Though communication protocols with the underlying framework are outside the scope of the manifest language, there must exist a correlation between the events generated by the monitors and the KPIs described in the manifest. This is modelled in Figure 3.

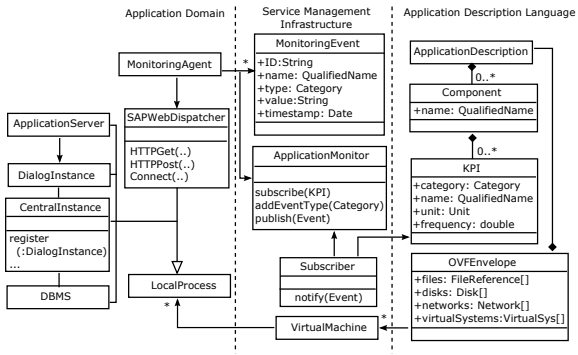

Fig. 3 Application Description Language

Based on our running example, the figure exemplifies the relationship between the ADL, the RESERVOIR application-level monitoring infrastructure, and the application domain. The syntax of the ADL consists of one or more named components, with a number of associated KPIs. These KPIs are identified using appropriate qualified names (e.g. `com.sap.webdispatcher.kpis.sessions`), that will allow the underlying infrastructure to identify corresponding events obtained from an application level monitor and forward these to subscribers responsible for the enforcement of elasticity rules.

We are concerned in the SAP example with the number of simultaneous web sessions managed by the web dispatcher, as there is a proportional relationship between resource requirements and sessions. The number of simultaneous sessions will be used as a basis for scaling the number of Dialog Instances.However, directly monitoring the traffic to and from the web dispatcher would be impossible, as SAP uses proprietary protocols. The SAP system can nonetheless report the number of sessions provided an appropriate query is formulated. The monitoring agent would be responsible for such queries and forwarding obtained responses, bridging the gap between application and monitoring infrastructure.

KPI qualified names would be considered global within the scope of a service. If there exists a need to distinguish the KPI measurements produced by multiple instances of a same component, this is achieved by using distinct qualified names. Monitoring agents can, for example, include instance IDs in the qualified name. The structure of the qualified name itself would not fall within the scope of the manifest specification. Instances of an application service as a whole however would be considered distinct. At the implementation level, KPIs published within a network are tagged with a particular service identifier, and rules, covered below, will also be associated with this same identifier. Multiple instances of an application service would hence operate independently.

Elasticity Rules With respect to the rule syntax, we adopt an *Event-Condition-Action* approach to rule specification. This is a widely adopted model for rule definition, adopted for example in active databases and rule engines, and suited in this instance. Based on monitoring events obtained from the infrastructure, particular actions from the VEEM are to be requested when certain conditions relating to these events hold true. This requires rules to be expressed with respect to the interface of the underlying VEEM and monitoring events.

A representation of the elasticity rules based on a general rule-base model and their relationship to monitoring events and the Cloud infrastructure is illustrated in Figure 4. The syntax specifies conditions, based on monitoring events at the application layer or otherwise, which would lead to specified actions, based on a set of operations presented by the VEEM. The operations, modelled on the OpenNebula framework capabilities will involve the submission, shutdown, migration, reconfiguration, etc. of VMs and should be invoked within a particular time frame. The conditions are expressed using a collection of nested expressions and may involve numerical values, arithmetic and boolean operations, and values of monitoring elements obtained. The relationship between KPIs specified in the manifest and these events has been described in the previous section. The elasticity rules will be supervised by the Cloud infrastructure at the Service Manager layer during the running of the software system and it is expected that a rule interpreter will receive events from the infrastructure or application monitors and trigger such operations accordingly.

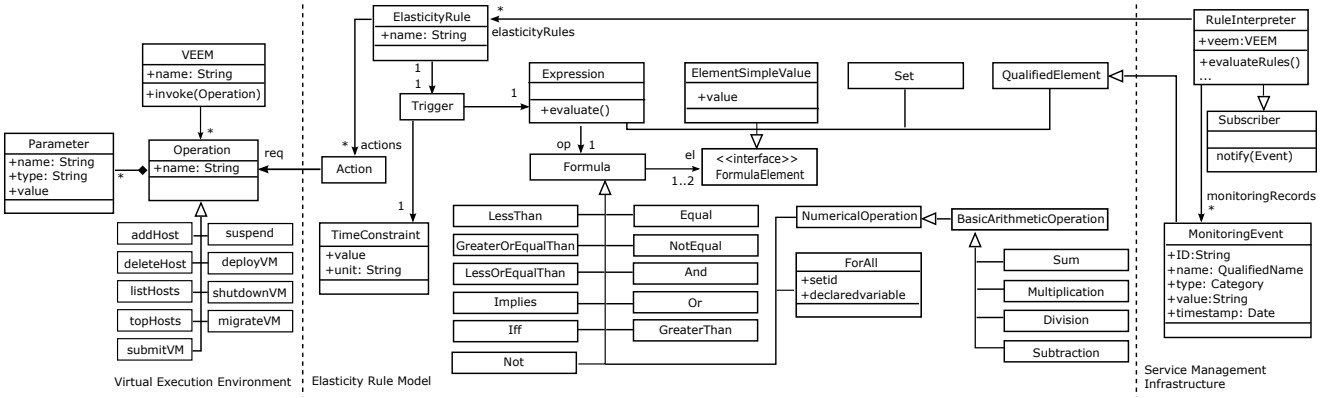

Fig. 4 Elasticity Rules Syntax

With respect to the example, this language enables us to express that virtual machines with new Dialog Instances should be created as the number of user sessions maintained by the SAP web dispatcher grows in order to handle the increased load. A concrete example of an elasticity rule will be provided in Section 6.

It is worth briefly discussing the subject of time management. The service provider controls the timeliness of the response in multiple ways. Firstly the rate at which monitoring events are sent by the application level monitor is entirely dictated by the application and this should be balanced against expected response time to avoid duplicate responses. Secondly service providers can specify a time frame within which particular actions should take place, as described above. Finally, the current time can be introduced as a monitorable parameter if necessary.

Additionally service providers may prefer expressing conditions regarding a series of measurements within a time frame rather than focusing on single events. We may be concerned here with the average number of active sessions in a window in order to limit the impact of strong fluctuations. While the language is extensible and presents the opportunity to provide such functionality, and we are currently working on the ability to specify a time series and operations related to that time series (mean, minimum, maximum, etc.), this can be achieved by aggregating measurements at the application level, with the monitoring agent performing such tasks.

Elasticity rules can be a powerful tool to express capacity constraints. The structure is kept purposely simple: not intended as a substitute for a programming language, elasticity rules only aim to inform the Cloud infrastructure of the corrective process to be undertaken. Auto-scaling is not a form of capacity planning but it aids in introducing a certain degree of flexibility in resource allocation which ensures that strong and often unexpected variations in demand can be met. In general, more complex relationships between performance indicators can be computed at the application level, before being forwarded to the service manager.

4.2.2 Semantic Definition

We examine in this section the dynamic semantics of the manifest language as OCL constraints on the relationship between the syntactic model of the manifest and the infrastructure and application domains. Dynamic semantics are concerned with deployment and run-time operation. These will specify behavioural constraints that the system should adhere to during its execution.

The question of how and when we verify that these constraints hold true during the provisioning of a service should be discussed briefly. Defined invariants should be true at any moment in time, however it is not feasible in practice to continuously check for this. Instead it is preferable to tie the verification to monitoring events or specific actions, such as a new deployment. Another question to be posed is what should be done when an evaluation of the state system does not fit the specified constraints. This will depend on the context: an exception may occur, or an operation should be invoked to trigger some corrective action, as would be the case with elasticity rules.

Service Deployment As the manifest is processed by the various independent components of the Service Manager to generate a deployment descriptor for submission to the VEEM, it becomes important to ensure that the final product, which may be expressed using a different syntax, is still in line with the requirements of the service provider. In the case of RESERVOIR, the VEEM would introduce it's own deployment template. Using OpenNebula as a reference implementation of a VEEM, the deployment template relied upon by the system is roughly based on a Xen configuration file. The association between manifest and deployment template is illustrated in Figure 5.

It is presumed that the service manager's `ManifestProcessor` will be responsible for parsing

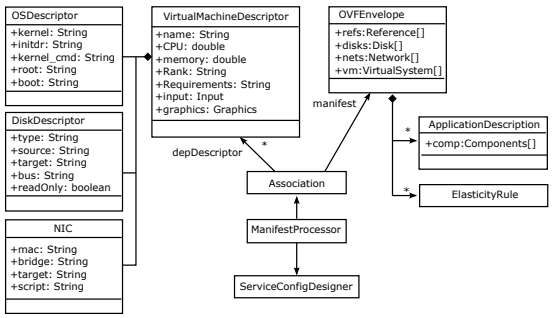

Fig. 5 Service Manifest and Deployment Descriptor

the manifest and generating one or more deployment templates accordingly. The `ServiceConfigAnalyzer` may be used to further optimise the placement with regards to the multiple sites at which it may be deployed, though the manifest specification is not concerned with this. It is only necessary to ensure that the optimisation process respects certain constraints regarding resource requirements. This is a *design by contract* approach [36]. We are not concerned with the actual transformation process, but rather that the final product, i.e. the deployment descriptor, respects certain constraints. These can be expressed in OCL as follows:

```
context Association
inv:
manifest.vm -> forAll ( v |
  depdescriptor.exists( d |
   d.name = v.id &&
   d.memory = v.virtualhardware.memory &&
   d.disk.source =
    (manifest.refs.file->asSet()->
      select(id = v.id))->first().href
   ...
   )
```

This OCL description is a sample of what is required to establish a relationship between manifest and deployment descriptor. Here, we describe that there should be at least one deployment descriptor generated for every virtual system described in the manifest definition that has the same identifier and memory requirements. The full OCL specification contains a full mapping of attributes of our manifest language to that of a VEEM deployment descriptor.

Service Elasticity Similarly, we can specify the expected outcome of elasticity rule enforcement with respect to both the syntax of the manifest and the underlying RESERVOIR components. OCL operations are *side effect free*, as in they do not alter the state of the system. Nevertheless they can be used to verify that the dynamic capacity adjustments have indeed taken place when elasticity rule conditions have been met, using the *post* context.

This is described in OCL as follows:

```
-- Collect monitoring records upon notification
context RuleInterpreter::notify(e: Event)
post: monitoringRecords =
        monitoringRecords@pre->append(e)
-- Evaluate elasticity rules and check adjustment
context RuleInterpreter::evaluateRules()
post: elasticityRules->forAll(er |
    if self.evaluate(er.expr) > 0 then
       er.actions->forAll(a | veem^invoke(a.req))
    else true
    endif)
-- Query simple type value
context RuleInterpreter::evaluate(el:
              ElementSimpleType):Real
post: result = el.value
-- Obtain latest value for monitoring record
-- with specific qualified name
context RuleInterpreter::evaluate(qe:
              QualifiedElement): Real
post:
if monitoringRecords->select(name=qe.name)
      ->last()->exists() then
   result = monitoringRecords
      ->select(name=qe.name)
      ->last().value
else result = qe.default
endif
-- Evaluate expressions
-- Defined as post in order to use recursion
context RuleInterpreter::evaluate(expr:
              Expression): Real
post:
 if expr.op.oclIsTypeOf(GreaterThan) then
     if self.evaluate((op.el->first()) >
        self.evaluate((op.el->last()) then
        result = 1
     else result = 0
     end if
 else
      ....
 endif
```

This OCL pseudo-code is only a subset of the complete OCL specification that aims to illustrate how we can specify the correct execution of elasticity rules with respect to the rule syntax. The code is split into a number of individual segments. The first simply states that monitoring events obtained are expected to be collected as records for the purpose of later evaluation. The second element states that if the conditional element of one of the elasticity rules is found to be true, then particular actions should have been invoked. To reiterate, the operations are side effect free, implying that no processing of any kind will take place in an OCL statement. Instead we only check that there has been communication with the VEEM to invoke certain operations if the conditions described hold true. How it is implemented is then left to developers.

The final segments relate to the evaluation of the conditions themselves. The `RuleInterpreter: evaluate(qe: QualifiedElement)` describes that upon evaluation of the rules, values for key performance indicators described in the document are obtained from the monitoring records, by examining the latest monitoring event with matching qualified name. This defines the relationship between KPIs and monitoring events. This asynchronous model is chosen because the Cloud infrastructure does not control application level monitoring agents. As there is no guarantee over how often monitoring information is provided, and rules may involve measurements from several services, it is for the implementation to determine when the rules should be checked to fit

within particular timing constraints rather than tying checks to the reception of any specific monitoring event. Finally the last segment illustrates the recursive evaluation of expressions based on the type of formula selected by the service provider.

4.2.3 Concrete syntax

While the specification of our manifest language is kept free of implementation concerns, the model-denotational approach adopted here provides a basis for automatically deriving concrete human or machine readable representations of the language that can be manipulated by the end-user or processed directly by the RESERVOIR based infrastructure. Moreover, beyond creating and editing the manifest itself, the syntax and accompanying semantics can be used as input for a generative programming tool to automate the generation of applications to control the provisioning process.

In practice, the RESERVOIR architecture may be implemented using a wide range of programming languages and existing technologies. The semantic definition described in this paper will generally serve as an important software engineering artefact, guiding the design and development of components. However, the potential for errors to occur during the provisioning process always exists, due to implementation or a failure to correctly interpret the specification of our language. We can assist in identifying and flagging such errors by *programatically* generating monitoring instruments which will validate run-time constraints previously described in Section 4.2.2. The process by which this is achieved is illustrated in Figure 6.

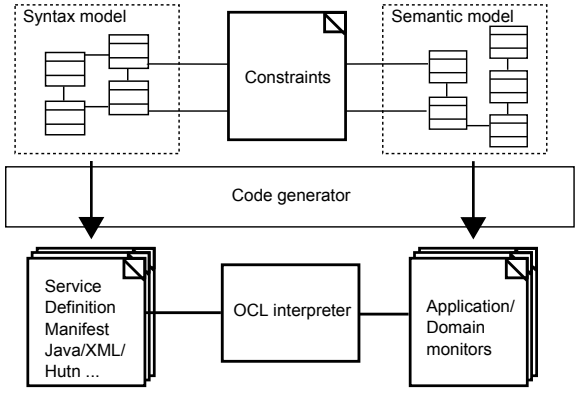

Fig. 6 Programatic Generation of Monitoring Instruments

In previous work, we have developed the UCL-MDA tools [28], a graphical framework for the manipulation of EMOF and OCL implemented as a plug-in for the Eclipse IDE. The framework relies on existing standards for the transformation of EMOF models and OCL into code, such as the Java Metadata Interface (JMI) standard and OCL interpreters, and is available at [34].

We have extended the framework for the purpose of this work. Our extensions introduce the ability to create, edit and validate manifests describing services to be deployed on a RESERVOIR based infrastructure. Element attribute values are input via the graphical interface in accordance with the structure of the language. Infrastructure related attributes and configuration values may be included in order to verify that OCL constraints are correctly maintained. This may be used amongst other things to verify that deployment descriptors generated by the infrastructure fit within the requirements specified by the manifest as covered.

Via the interface, users can additionally request the creation of stand-alone monitoring instruments in Java capable of interaction with our implementation of the RESERVOIR framework, which will be described in detail in Section 5. These are currently of two forms. The first is simply responsible for gathering and reporting the values of specific KPIs described in the manifest. The second will validate the correct enforcement of elasticity rules by evaluating incoming monitoring events and verifying where appropriate that suitable adjustment operations were invoked by matching entries and time frames in infrastructural logs. The framework also allows the generation of custom stubs which the service provider may used as a basis for the development by the service provider of monitoring agents, handling issues such as communication protocols, measurement labelling and packaging, and providing a control interface to manage frequency and operation. This would have to be supplemented with appropriate probes responsible for the application level collection of measurements.

Java code is generated from a combination of data obtained from the specification, element values input by the user and Java templates, the latter being used to bridge the gap between the abstract model of the infrastructure and the actual implementation. As previously discussed, issues such as communication channels for the distribution of monitoring events fall outside the scope of the manifest language specification. Templates provide the necessary code to gather KPI measurements or parse infrastructure logs and pass this information to OCL interpreters.

The tool hence serves the following purposes: firstly, it allows users to specify and manipulate manifests. Secondly it allows the generation of code allowing the service provider to verify the correct provisioning of a service at run-time according to the semantics of the language. Finally it provides the means of interfacing a service with the RESERVOIR monitoring architecture.

5 Service Lifecycle Management

The service lifecycle encapsulates the initial deployment of a service, whereby a Service Provider sends a manifest described in the language detailed above to a cloud, through to the first instantiation of one or more service components, the monitoring of said components, and finally the additional deployment, undeployment and resizing of service components as demand and workload evolves. All of these complex functions are undertaken through the collaboration of the Service Manager, the Monitoring Framework, and the VEEM, and are just one of the many control loops that are required within a cloud computing environment.

In this section we describe these components and discuss the implementation decisions that were taken to ensure a scalable, manageable service management infrastructure.

5.1 Service Manager

Previous sections have dealt with the language to define service manifests, describing the requirements that have driven its design along with its abstract and concrete syntax. In the current subsection we describe in detail the Service Manager, which is the main component of the RESERVOIR middleware that processes descriptions in the manifest language, instantiates the services, and manages these services throughout their lifecycles.

The Service Manager within the overall RESERVOIR stack has been already introduced in Figure 1, with a more detailed architecture shown in Figure 7. It exposes a deployment interface to Service Providers, based on the OVF-based service manifest language discussed in Section 4.2.1. It interacts with the Virtual Execution Environment Manager (VEEM) via a REST based interface to handle the deployment and management of the virtual machines composing the application of the service providers .

The main components of the Service Manager are described below. Although it includes many additional components (e.g. for accounting and billing), we focus here on the ones related to service deployment and elasticity for the sake of clarity. These are:

Manifest parser: The parser handles and processes the service specification (in OVF) provided by the Service Provider, extracting from it a suitable service lifecycle that meets the provider requirements.

Service Lifecycle Manager: This component controls the service lifecycle and is in charge of all service management operations, including initial deployment, runtime scaling and service termination. The Service Lifecycle Manager orchestrates all the other Service Manager components and interfaces with the VEEM in order to actually implement the management operations, e.g. sending individual deployment descriptors to create new VEEs.

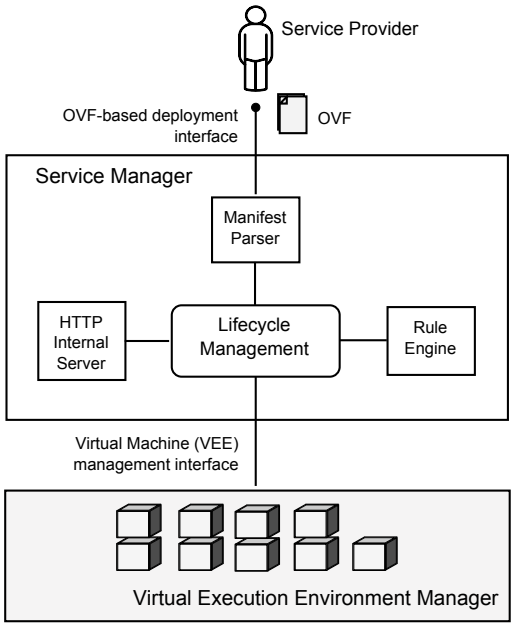

Fig. 7 Service Manager Architecture

Rule Engine: The rule engine enforces scaling rules during service runtime. It is based on a business engine (the current Service Manager implementation uses Drools [11]) which takes KPI monitoring information as input for the given rule set derived from our specification, resulting in scaling operations when some rule is triggered.

HTTP internal server: The Service Manager includes an internal HTTP server to provide virtual machine images that the VEEM needs to deploy new virtual machines. In particular, the HTTP server is used to place both the base image containing operating system and service software and the image containing the customization data. The server is needed because it is preferable to include references to the images in the REST messages than passing the actual images themselves, which are usually very large.

Taking into account these various components, we now describe how the Service Manager implements the manifest language semantics descibed in Section 4.2.2 for both service deployment and service elasticity.

5.1.1 Service Deployment

The service deployment process consists of 7 steps. These are illustrated in the left of Figure 8. The workflow is as follows:

(1) The Service Provider issues a service deployment operation to the Service Manager. The main parameter for

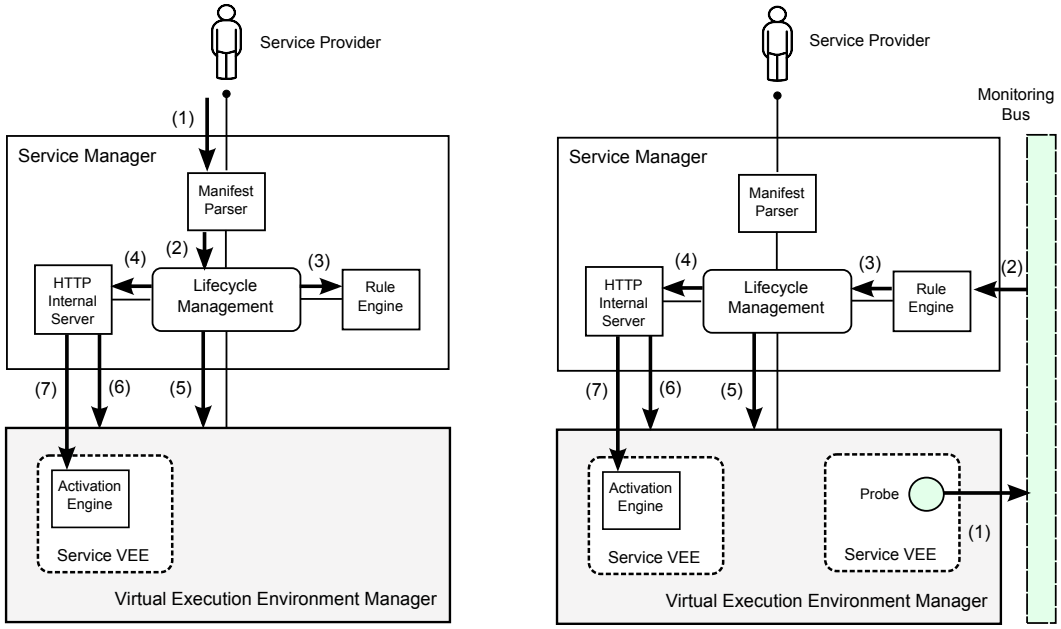

Fig. 8 Service Deployment Workflow (left) and Service Elasticity Workflow (right)

this operation is the service manifest, expressed in OVF. The Manifest Parser processes the file and, as result of this task, an internal representation of the service manifest is built, to be used for the other Service Manager components.

(2) The service deployment command is issued to the Service Lifecycle Manager.

(3) The Service Lifecycle Manager sets up and installs the elasticity rules specified in the manifest in the Rule Engine, so it starts enforcing them when the service gets deployed.

(4) The Service Lifecycle Manager interacts with the HTTP server to set up all the images required by the virtual machines composing the service. For each virtual machine, two images are provided – the base image, and the disk containing the customisation data (e.g. IP address) according to the OVF Environment format (see [10]). Both the reference to the base image and the customisation data are extracted from the service manifest.

(5) The Service Lifecycle Manager sends a deployment descriptors to the VEEM to create a new VEE.

(6) The VEEM gets the base disk for the VEE, creates it and boots it. The created VEE is shown in a dashed line in the figure.

(7) The customization disk is attached to the VEE (typically as a virtual CD/DVD) so the Activation Engine that runs as part of the VEE boot procedure can access the customization data and configure the VEE properly (e.g. setting the assigned IP in the operating system configuration).

It is worth mentioning that, although we are showing just the creation of a single virtual machine, the subflow composed by steps 5-to-7 repeat for every VEE needed for initial service deployment. For example, if the initial layout of the service is composed of a load balancer, a web server, and a database, each with its own virtual machine, then three of these 5 to 7 cycles will be done.

5.1.2 Service Elasticity

When considering a running service, the process by which elasticity is managed has a dedicated workflow, which is shown in the right hand side of Figure 8. The steps are as follows:

(1) Monitoring probes running in the virtual machines are continuously sending KPI measures to the Monitoring Framework.

(2) The Monitoring Framework provides KPI information to the Rule Engine running at the Service Manager level.

(3) When the KPI value triggers a given elasticity rule to scale up the service, the Rule Engine issues a command to the Service Lifecycle Manager.

(4) A customization disk is generated for that image in the HTTP internal server.

(5) The Service Lifecycle Manager follows the conventional procedure (described in Section 5.1.1 workflow in steps 5-to-7) to deploy a new virtual machine.

5.2 Monitoring Framework

Key to facilitating the ow of information in a cloud based environment is a monitoring process via which various metrics regarding the operation of services and infrastructure can be circulated to the required components in a scalable and effective way. We have already covered the relationship between provider requirements specified in the manifest and measurements obtained from the monitoring framework, and resulting scaling operations.

Here we discuss the implementation of the monitoring framework itself, and the design decisions that were taken to ensure that its operation does not affect the performance of the network itself or the running service applications. This is achieved by ensuring that the management components only receive data that is of relevance: In a large distributed system there may be hundreds or thousands of measurement probes which can generate data. It would not be effective to have all of these probes sending data all of the time, so a mechanism is needed that controls and manages the relevant probes.

Existing monitoring systems such as Ganglia [17], Nagios [19], MonaLisa [20], and GridICE [2] have addressed monitoring of large distributed systems. They are designed for the fixed, and relatively slowly changing physical infrastructure that includes servers, services on those servers, routers and switches. However, they have not addressed or assumed a rapidly changing and dynamic infrastructure as seen in virtual environments. In the physical world, new machines do not appear or disappear very often. Sometimes some new servers are purchased and added to a rack, or a server or two may fail. Also, it is rare that a server will move from one location to another. In the virtual world, the opposite is the case. Many new hosts can appear and disappear rapidly, often within a few minutes. Furthermore, the virtual hosts, can be migrated from one network to another, still retaining their capabilities.

It is these characteristics that provide a focus for the monitoring framework. We have determined that the main features for monitoring in a virtualized environment which need to be taken account of are:

Scalability: to ensure that the monitoring can cope with a large numbers of probes

Elasticity: so that virtual resources created and destroyed by expanding and contracting services are monitored correctly

Migration: so that any virtual resource which moves from one physical host to another is monitored correctly

Adaptability: so that the monitoring framework can adapt to varying computational and network loads in order to not be invasive

Autonomic: so that the monitoring framework can keep running without intervention and reconfiguration

Federation: so that any virtual resource which reside on another domain is monitored correctly

To establish such features in a monitoring framework requires careful architecture and design.

The RESERVOIR monitoring system covers all of the layers and components presented in Figure 1 of Section 2. The following sections provide details of the design of the RESERVOIR monitoring system.

5.2.1 Producers and Consumers

The monitoring system itself is designed around the concept of producers and consumers. That is there are producers of monitoring data, which collect data from probes in the system, and there are consumers of monitoring data, which read the monitoring data. The producers and the consumers are connected via a network which can distribute the measurements collected.

The collection of the data and the distribution of data are dealt with by different elements of the monitoring system so that it is possible to change the distribution framework without changing all the producers and consumers. For example, the distribution framework can change over time, say from IP multicast, to an event bus, or a publish / subscribe framework. This should not affect too many other parts of the system.

5.2.2 Data Sources and Probes

In many systems probes are used to collect data for system management [17] [9]. In this regard, this monitoring framework will follow suit. However, to increase the power and flexibility of the monitoring we introduce the concept of a data source. A data source represents an interaction and control point within the system that encapsulates one or more probes. A probe sends a well defined set of attributes and values to the consumers, defined in a data dictionary. This can be done by transmitting the data out at a predefined interval, or transmitting when some change has occured.

The measurement data itself is sent via a distribution framework. These measurements are encoded to be a small as possible in order to maximise the network utilization. Consequently, the measurement meta-data is not transmitted each time, but is kept separately in an information model. This information model can be updated at key points in the lifecycle of a probe and can be accessed as required by consumers.

5.2.3 Probe Data Dictionary

One of the important aspects of this monitoring design is the specification of a Data Dictionary for each probe. The Data Dictionary defines the attributes as the names, the types and

the units of the measurements that the probe will be sending out. These consist essentially of the KPIs that are specified in the service manifest.

This is important because the consumers of the data can collect this information in order to determine what will be received. In particular, for the Service Manager, this information will allow specifications in the manifest to be checked in elasticity and SLA rules. At present many monitoring systems have fixed data sets, with a the format of measurements being pre-defined. The advantage here is that as new probes are to added to the system or embedded in the application, it will be possible to introspect what is being measured.

The measurements that are sent will have value fields that relate directly to the data dictionary. To determine which field is which, the consumer can lookup in the data dictionary to elaborate the full attribute value set. Probes will be embedded in both the infrastructure and the application service components themselves. To manage scaling according to our process, it will be the responsibility of the service provider to ensure that probes embedded in the virtual machines to be deployed rely on a data dictionary that is consistent with the KPIs specified in the service manifest.

5.2.4 Measurements

The actual measurements that get sent from a probe will contain the attribute-value fields together with a type and a timestamp, plus some identification fields. The attribute-values contain the information the probe wants to send, the type indicates what kind of data it is, and the timestamp has the time that the data was collected, as modelled in Section 4.2.1.

The identification fields are used to determine for which component or which service and from which probe this data has arrived from. We rely for this purpose on the qualified names discussed in Section 4.2.1. As there are multiple components which need monitoring and multiple running services, the consumer of the data must be able to differentiate the arriving data into the relevant streams of measurements.

5.2.5 Distribution Framework

In order to distribute the measurements collected by the monitoring system, it is necessary to use a mechanism that fits well into a distributed architecture such as the management overlay. We need a mechanism that allows for multiple submitters and multiple receivers of data without having vast numbers of network connections. For example, having many TCP connections from each producer to all of the consumers of the data for that producer would create a combinatorial explosion of connections. Solutions to this include IP multicast, Event Service Bus, or publish/subscribe mechanism.

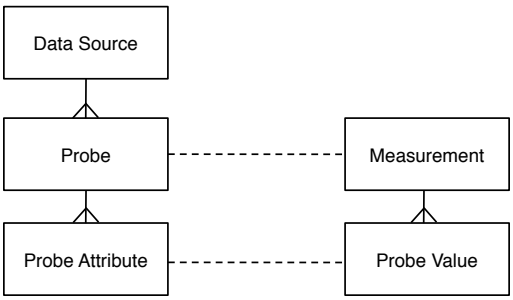

Fig. 9 Relationship Model

In each of these, a producer of data only needs to send one copy of a measurement onto the network, and each of the consumers will be able to collect the same packet of data concurrently from the network.

5.2.6 Design and Implementation Overview

Within the monitoring framework there are implementations of the elements presented in the relationship model shown in figure 9. In this model we see, a DataSource which acts as the control point and a container for one or more Probes. Each Probe defines the attributes that it can send. These are set in a collection of ProbeAttribute objects, that specify the name, the type, and the units of each value that can be sent within a measurement.

When a Probe triggers a monitoring event by sending a Measurement, the Measurement has a set of values called Probe Values. The Probe Values that are sent are directly related to the Probe Attributes defined within the Probe.

When the system is operating, each Probe reports the collected measurement to the Data Source. The Data Source passes these measurements to a networking layer, where they are encoded into an on-the-wire format, and then sent over the distribution network. The receiver of the monitoring data decodes the data and passes reconstructed Measurements to the monitoring consumer. Encoding measurement data is a common function of monitoring systems [17] as it increases speed and decreases network utilization.

In the monitoring framework, the measurement encoding is made as small as possible by only sending the values for a measurement on the data distribution framework. The definitions for the Probe Attributes, such as the name and the units are not transmitted with each measurement, but are held in the information model and are accessed as required.

The current implementation is written in Java, and the output for each type currently uses XDR [30]. As such each type defined uses the same byte layout for each type as defined in the XDR specification. All of this type data is used by a measurement decoder in order to determine the actual type and size of the next piece of data in a packet.

Key	Value
/datasource/datasource-id/name	datasource name
/probe/probe1-id/datasource	datasource-id
/probe/probe2-id/datasource	datasource-id
...	...
/probe/probeN-id/datasource	datasource-id

Table 1 Information Model Entries for a Datasource

Key	Value
/probe/probe-id/name	probe name
/probe/probe-id/datarate	probe data rate
/probe/probe-id/on	is the probe on or off
/probe/probe-id/active	is the probe active or inactive
/schema/probe-id/size	no of attributes N
/schema/probe-id/0/name	name of probe attribute 0
/schema/probe-id/0/type	type of probe attribute 0
/schema/probe-id/0/units	units for probe attribute 0
/schema/probe-id/1/name	name of probe attribute 1
...	...
/schema/probe-id/N/units	units for probe attribute 0

Table 2 Information Model Entries for a Datasource

5.2.7 Information Model Encoding

The Information Model for the Monitoring System holds all of the data about Data Sources, Probes, and Probe Data Dictionaries present in a running system. As Measurements are sent with only the values for the current reading, the meta-data needs to kept for lookup purposes. By having this Information Model, it allows consumers of measurements to lookup the meaning of each of the fields.

In many older monitoring systems this information model is stored in a central repository, such as an LDAP server. Newer monitoring systems use a distributed approach to holding this data, with MonAlisa using JINI as its information model store.

For the implementation of the Information Model we have used a Distributed Hash Table (DHT) for the distributed information model. This allows the receivers of Measurement data to lookup the fields received to determine their names, types, and units. The information model nodes uses the DHT to interact among one another.

The implementation has a strategy for converting an object structure into a path-based taxonomy for use as keys in the DHT. The IDs of the Data Sources and the IDs of the Probes are important elements of this taxonomy.

For each Data Source, the keys and values shown in table 1 are added to the DHT. For each Probe, the keys and values shown in table 2 are added to the DHT. For each Probe, the keys and values shown in table 2 are added to the DHT.

Using the encoded data from the information model, any of the consumers of the monitoring data, in particular the management overlay, can evaluate all the meta-data of the measurements.

The RESERVOIR monitoring system has been used successfully to provide data on all the elements of the cloud architecture and the running services [8] [16] [7]. The measurements supplied have been used for the service lifecycle management, such as service elasticity, as well as for accounting and billing. In the following section on evaluation, these measurements are used for the real execution of a computational chemistry application.

6 Experimental Evaluation

In the evaluation of our work we aim to prove the following hypothesis: provided that an architecture definition correctly specifies requirements and elasticity rules, and that the Cloud computing infrastructure obeys the constraints identified in the semantic definition, then the *quality of service* that can be obtained from a Cloud computing infrastructure should be equivalent to that obtained were the application hosted on dedicated resources. In addition, through the specification of elasticity rules, providers can considerably reduce expenditure by minimising over-provisioning.

We will demonstrate this hypothesis by deploying a production level service on an infrastructure consisting of the RESERVOIR stack. The Service Manager of the RESERVOIR stack incorporates monitors that validate the specified constraints. The selected service is a grid based application responsible for the computational prediction of organic crystal structures from the chemical diagram [12].

The application operates according to a predefined workflow involving multiple web based services and Fortran programs. Up to 7200 executions of these programs may be required to run, as batch jobs, in both sequential and parallel form, to compute various subsets of the prediction. Web services are used to collect inputs from a user, coordinate the execution of the jobs, process and display results, and generally orchestrate the overall workflow. The actual execution of batch jobs is handled by Condor [33], a job scheduling and resource management system, which maintains a queue of jobs and manages their parallel execution on multiple nodes of a cluster.

This case study provides many interesting challenges when deployed on a Cloud computing infrastructure such as RESERVOIR. Firstly, the application consists of a number of different components with very different resource requirements, which are to be managed jointly. Secondly, the resource requirements of the services will vary during the lifetime of the application. Indeed, as jobs are created, the number of cluster nodes required to execute them will vary. Our goal in relying upon a Cloud computing infrastructure will be to create a *virtualised cluster*, enabling the size of the cluster to dynamically grow and contract according to load.

For this evaluation, we will compare the quality of service, i.e. the duration required to complete the prediction)

when executing this workflow on a dedicated cluster, compared to a Cloud computing infrastructure that provides support for our abstractions. We are not concerned here with the overhead of hypervisors such as Xen, which are well documented [6]. Instead we are concerned with evaluating the costs of dynamically adjusting the resource provisioning during the application lifecycle and determining whether an appropriate level of service can still be obtained.

6.1 Testbed Architecture

6.1.1 Service components

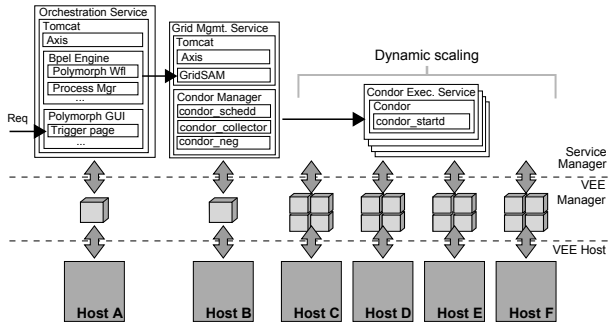

Fig. 10 Testbed Architecture

The testbed we use is illustrated in Figure 10. Three main types of service components can be distinguished. The *Orchestration Service* is a web based server responsible for managing the overall execution of the application. It presents an HTTP front end enabling users to trigger predictions from a web page, with various input parameters of their choice. The Business Process Execution Language (BPEL) [21], is used to coordinate the overall execution of the polymorph search, relying on external services to generate batch jobs, submit the jobs for execution, process the results and trigger new computations if required.

The *Grid Management Service* is responsible for coordinating the execution of batch jobs. It presents a web service based interface for the submission of jobs. Requests are authenticated, processed and delegated to a Condor scheduler, which will maintain a queue of jobs and manage their execution on a collection of available remote execution nodes. It will match jobs to execution nodes according to workload and other characteristics (CPU, memory, etc.). Once a target node has been selected it will transfer binary and input files over and remotely monitor the execution of the job.

The last type of component is the *Condor Execution Service*, which runs the necessary daemons to act as a Condor execution node. These daemons will advertise the node as an available resource on which jobs can be run, receive job details from the scheduler and run the jobs as local processes. Each node runs only a single job at a time and upon completion of the job transfers the output back to the scheduler, and advertises itself as available.

6.1.2 Deployment

Packaged as individual virtual machines encapsulating operating system and other necessary software components, the three components are deployed on the RESERVOIR-based infrastructure. The associated manifest describes the capacity requirements of each component, including CPU and memory requirements, references to the image files, starting order (based on service components dependencies), elasticity rules and customisation parameters. For the purpose of the experiment, the Orchestration and Grid Management Services will be allocated a fixed set of resources, with only a single instance of each being required. The Condor execution service however will be replicated as necessary, in order to provide an appropriate cluster size for the parallel execution of multiple jobs.

The elasticity rules will tie the number of required Condor execution service instances to the number of jobs in queue as presented by the Condor scheduler. This enables us to dynamically deploy new execution service instances as the number of jobs awaiting execution increases. Similarly as the number of jobs in the queue decreases it is no longer necessary to use the resources to maintain a large collection of execution nodes, and hosts in the Cloud can be released accordingly. This is expressed as follows in the manifest using an XML concrete syntax, which conforms to the abstract syntax described in Section 4.2.1. We use a similar elasticity rule for downsizing allocated capacity as the queue size shrinks.

```
<ElasticityRule name="AdjustClusterSizeUp">
 <Trigger>
  <TimeConstraint unit="ms">5000</TimeConstraint>
  <Expression>
   (@uk.ucl.condor.schedd.queuesize /
   (@uk.ucl.condor.exec.instances.size +1) > 4) &&
   (@uk.ucl.condor.exec.instances.size < 16 )
  </Expression>
  </Trigger>
  <Action run=
   "deployVM(uk.ucl.condor.exec.ref)"/>
<ElasticityRule>
```

The elasticity rules will refer to key performance indicators that are declared within the context of the application structure. This is expressed as follows:

```
<ApplicationDescription
 name="polymorphGridApp">
 <Component name="GridMgmtService" ovf:id="GM">
  <KeyPerformanceIndicator category="Agent" type="int">
   <Frequency unit="s">30</Frequency>
   <QName>uk.ucl.condor.schedd.queuesize</QName>
  </KeyPerformanceIndicator>
 </Component>
  ...
</ApplicationDescription>
```

All components and KPIs are declared in this manner. This enables the infrastructure to monitor KPI measurements being published by specific components and associate them to the declared rules according to the previously stated semantics. In this particular instance we are specifying that a monitoring agent associated with the Grid Management Service will publish measurements under the `uk.ucl.condor.schedd.queuesize` qualified name every 30 seconds as integers.

The overall management process can hence be described as follows: upon submission of the manifest, the Service Manager, which is responsible for managing the joint allocation of service components and service elasticity, will parse and validate the document, generating suitable individual deployment descriptors to be submitted to the VEEM beginning with the Orchestration and Grid Management components. The VEEM will use these deployment descriptors to select a suitable physical host from the pool of known resources. These resources are running appropriate hypervisor technology, in this case the Xen virtualisation system, to provide a virtualised hardware layer from which the creation of new virtual machines can be requested. Upon deployment, the disk image is replicated and the guest operating system is booted with the appropriate virtual hardware and network configuration.

When the Grid Management component is operational, a monitoring agent, as described in Section 4.2.1, will begin the process of monitoring the queue length and broadcast the number of jobs in the queue on a regular basis (every 30 seconds) under the selected qualified name (`uk.ucl.condor.schedd.queuesize`). These monitoring events, combined with appropriate service identifier information, will be recorded by the rule interpreter component of the Service Manager to enforce elasticity rules. When conditions regarding the queue length are met (i.e. there are more than 4 idle jobs in the queue), the Service Manager will request the deployment of an additional Condor Execution component instances. Similarly, when the number of jobs in queue falls below the selected threshold, it will request the deallocation of virtual instances.

The actual physical resources which are managed by the RESERVOIR infrastructure used in this experiment consist of a collection of six servers, each of them presenting a Quad-Core AMD Opteron(tm) Processor 2347 HE CPU and 8 GBs of RAM and with shared storage via NFS. OpenNebula v1.2, as the VEEM implementation, is used to manage the deployment of virtual machines on these resources according to the requirements specified by a Service Manager.

Both the Orchestration and Grid Management components will be allocated the equivalent of a single physical host each, due to heavy memory requirements, and up to 4 Condor Execution components may be deployed on a single physical host, limiting the maximum cluster size to 16 nodes. This mapping however is transparent to the Service Manager, Service Provider and application.

6.1.3 Metrics

It is also important to briefly describe the characteristics of the overall application workflow, in order to determine appropriate metrics for the experiment. Our primary indicator of quality of service is the overall *turn around time* of a prediction. The turn around time can be defined as the amount of time elapsed between the moment a client user requests a search to the moment results are displayed on the web page. As previously stated, the overall process combines functionality from a number of different Fortran programs into a larger workflow. Based on our selected input, two long running jobs will first be submitted, followed by an additional set of 200 jobs being spawned with each completion to further refine the input. We must also take into account the additional processing time involved in orchestrating the service and gathering outputs.

Another important metric to consider is that of resource usage. The goal of service elasticity is to reduce expenditures by allowing Service Providers to minimise overprovisioning. While the actual financial costs will be dependent on the business models employed by Cloud infrastructure providers, we can at the very least rely upon resource usage as an indicator of cost.

6.1.4 Experiment results

We compare turn-around time and resource usage obtained on our Cloud infrastructure with elasticity support with that obtained in an environment with dedicated physical resources. The objective is to verify that there are no strong variations in turn around time, but a significant reduction in resource usage. The results are illustrated in Figure 11. The number of queued jobs is plotted against the number of Condor execution instances deployed. Both charts show large increases in queued jobs as the first long running jobs complete and the larger sets are submitted. In addition, the first chart represents the execution of the application in a dedicated environment and shows a set of 16 continuously allocated execution nodes. The second chart represents the execution of the application with elasticity support, shows the increase in the number of allocated nodes as jobs in queue increases, and a complete deallocation as these jobs complete. The overall turn around time and resource usage obtained is described in Table 3.

As we can see from the results, a 7.15% increase in turn around time occurs. As there is little difference in execution times in the individual batches of jobs on either the dedicated or virtual cluster, the increase in turn around time comes primarily from the additional time that is taken to

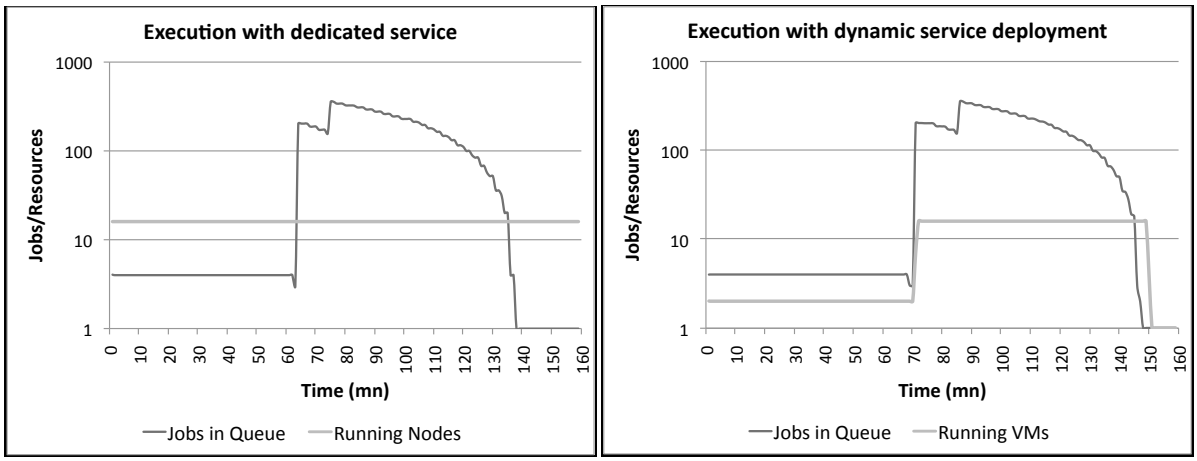

Fig. 11 Job Submission and Resource Availability

	Dedicated Environment	Cloud Infrastructure
Search turn around time (s)	8605	9220
Complete shutdown time (s)	N/A	9574
Average execution nodes		
for run	16	10.49
until shutdown	N/A	10.42
Percentage differences		
Resource usage saving		34.46%
Extra run time (jobs)		7.15%

Table 3 Experiment Results

create and deploy new instances of the Condor execution service as jobs are added in the queue. This can be verified in Figure 11, where a small delay can be observed between increases in the number of jobs in queue, and the increase in Condor execution services. The overhead incurred is due to the deployment process, which will involve duplicating the disk image of the service, deploying it on a local hypervisor, and booting the virtual machine, and the registration process, which is the additional time required for the service to become fully operational as the running daemons register themselves with the grid management service. There exists ways of reducing this overhead independently of the Cloud computing infrastructure, at the expense of resource usage, such as relying on pre-existing images to avoid replication.

A 10 minute increase of time in can however be constituted as reasonable considering the overall time frame of a search, which is well over 2 hours. This is particularly true as we consider the overall resource usage savings. Indeed as can be seen in the table, with respect to execution nodes, the overall resource usage decreases by 34.46% by relying on service elasticity. This is because the totality of the execution nodes are not required for the initial bulk of the run, where only 2 jobs are to be run. It is only in the second stage that more nodes are required to handle the additional jobs.

Of course the savings here are only considered in the context of the run itself. If we consider the overall use of the application over the course of a randomly selected week on a fully dedicated environment where resources are continuously available, even more significant cost savings will exist. Examining logs of searches conducted during this period, and based on cost savings obtained here, we have estimated that overall resource consumption would drop by 69.18%, due to the fact that searches are not run continuously; no searches were run on two days of the week, and searches, though of varying size, were run only over a portion of the day, leaving resources unused for considerable amounts of time.

7 Related Work

Much of this work builds on the foundation previously established with SLAs, where we used a model denotational approach to specify service level agreements for web-based application services [29]. In this paper, we have aimed to broaden the approach to encapsulate Cloud computing primitives and environment, providing a specification for a manifest language describing software architecture, physical requirements, constraints and elasticity rules.

In addition, it is worth examining research developments related to service virtualisation, grid computing and component based software architecture description languages. With respect to virtual environment deployment descriptions, the manifest language proposed here builds upon the Open Virtualisation Format (OVF) [10], whose limitations have already been discussed.

There exists a number of software architecture description languages which serve as the run-time configuration and deployment of component based software systems. The CDDLM Component Model [32], for example, outlines the requirements for creating a deployment object responsible for the lifecycle of a deployed resource with focus on grid services. Each deployment object is defined using the CDL language. The model also defines the rules for managing the

interaction of objects with the CDDML deployment API. Though the type of deployment object is not suited to virtual machine management, the relationship between objects and the deployment API can be compared to our approach we have undertaken here, providing a semantic definition for the CDL language. However in our case the relationship between domain and syntactic models is performed at a higher level of abstraction, relying on OCL to provide behavioural constraints. Our specification is hence free of implementation specific concerns.

The general approach to dynamic and automated provisioning may also be compared to the self-managing computing systems associated with autonomic computing research [14]. While our approach to elasticity is *explicit*, in that providers define appropriate scaling rules based on an event condition action model, we have laid a foundation for further methods to be developed relying on predictive and autonomic mechanisms to anticipate future allocation changes and further minimise over-provisioning, providing monitoring channels and a rule based framework for the dynamic management of services.

Finally it is important to examine current developments in production level Cloud environments, such as Amazon's EC2 offering [1]. In particular, auto-scaling has been introduced by Amazon to allow allocated capacity to be automatically scaled according to conditions defined by a service provider. These conditions are defined based on observed resource utilisation, such as CPU utilisation, network activity or disk utilisation. Whilst the approach laid out in this paper can be used to define elasticity rules based on such metrics, this can prove limiting. With respect to the evaluation, the need to increase the cluster size cannot be identified through these metrics as we require an understanding of the scheduling process. The ability to describe and monitor application state is crucial if we wish to correctly anticipate demand.

In addition, the focus of our paper has primarily been on Infrastructure-as-a-Service Clouds. Nevertheless it is still important to briefly discuss the relevance of this work with respect to Platform-as-a-Service (PaaS) Clouds such as Windows Azure [18]. PaaS Clouds provide an additional level of abstraction over IaaS Clouds, providing a runtime environment for the execution of application code and a set of additional software services, such as communication protocols, access control, persistence, etc. Windows Azure allows services to be described as distributed entities: clients can specify the interfaces exposed by services, communication end points, channels and roles (web or worker) and different hardware requirements may be allocated. However a need to control the management, distribution and lifecycle of multi-component systems still exists, though with the added benefit of application specific operations being more readily exposed to the infrastructure. How this is implemented will be tied to the specifics of the platform itself but we do believe there is potential to adapt many aspects of our approach to the platform specific interfaces and tools.

8 Conclusion and Future Work

In this paper, we have proposed an abstract syntax and semantic definition for a service manifest language which builds on the OVF standard and enables service requirements, deployment constraints (placement, co-location and startup/stopping order) and elasticity rules to be expressed. We believe that clear behavioural semantics are of paramount importance to meet quality goals of both the Service and Infrastructure provider. Our model-driven approach aims to strengthen the design of the RESERVOIR stack, identifying functional capabilities that should be present and constraints the system should observe. We also explored the relationship between a novel RESERVOIR monitoring infrastructure and service manifest, focusing particularly on notions such as data dictionary and the KPIs defined at the abstract layer. Such relationship has served to drive the implementation of the monitoring infrastructure.

We have shown experimentally that the implementation of these concepts is feasible and that a complete architecture definition that uses our manifest syntax can enable Cloud computing infrastructure to realise significant savings in resource usage, with little impact on overall quality of service. Given that a Cloud can deliver a near similar quality of service as a dedicated computing resource, Clouds then have a substantial number of advantages. Firstly application providers do not need to embark on capital expenditure and instead can lease the infrastructure when they need it. Secondly, because the elasticity rules enable the application provider to flexibly expand and shrink their resource demands so they only pay the resources that they actually need. Finally, the Cloud provider can plan its capacity more accurately because it knows the resource demands of the applications it provides.

While Cloud computing is still a relatively new paradigm, and as such changes in standards, infrastructural capabilities and component APIs are inevitable, defining the software architecture of services hosted on a Cloud with respect to the capabilities of the underlying infrastructure is key to optimizing resource usage. This allows us to bridge the gap between application and infrastructure and provides the means for providers to retain some control over the management process.

Our work paves the way towards quality of service aware service provisioning. In future work, we aim to develop appropriate syntax and semantics for resource provisioning service level agreements. Building upon the approach laid out here, we aim to provide a framework for the automated monitoring and protection of service level obligations based on defined semantic constraints.

References

1. Amazon (2006) Amazon Elastic Compute Cloud (Amazon EC2). [Online] http://aws.amazon.com/ec2
2. Andreozzi S, De Bortoli N, Fantinel S, Ghiselli A, Rubini GL, Tortone G, Vistoli MC (2005) GridICE: A monitoring service for grid systems. Future Gener Comput Syst 21(4):559–571, DOI http://dx.doi.org/10.1016/j.future.2004.10.005
3. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
4. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM Press, New York, NY, USA, pp 164–177, DOI http://doi.acm.org/10.1145/945445.945462
5. Chapman C, Emmerich W, Márquez FG, Clayman S, Galis A (2010) Software Architecture Definition for On-demand Cloud Provisioning. In: 19th ACM International Symposium on High Performance Distributed Systems, ACM
6. Cherkasova L, Gardner R (2005) Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In: Proceedings of the USENIX annual technical conference, pp 387–390
7. Clayman S, Galis A, Chapman C, Toffetti G, Rodero-Merino L, Vaquero L, Nagin K, Rochwerger B (2010) Monitoring service clouds in the future internet. In: Towards the Future Internet - Emerging Trends from European Research, IOS Press
8. Clayman S, Galis A, Mamatas L (2010) Monitoring virtual networks with lattice. In: Management of Future Internet - ManFI 2010, URL http://www.manfi.org/2010/
9. Cooke A, Gray AJG, Ma L, Nutt W, et al (2003) R-GMA: An information integration system for grid monitoring. In: Proceedings of the 11th International Conference on Cooperative Information Systems, pp 462–481
10. DMTF (2009) Open Virtualization Format. Specification DSP0243 v1.0.0, Distributed Management Task Force
11. DROOLS (2010) DROOLS. [Online] http://jboss.org/drools
12. Emmerich W, Butchart B, Chen L, Wassermann B, Price SL (2005) Grid Service Orchestration using the Business Process Execution Language (BPEL). Journal of Grid Computing 3(3-4):283–304, URL http://dx.doi.org/10.1007/s10723-005-9015-3
13. Galán F, Sampaio A, Rodero-Merino L, Loy I, Gil V, Vaquero LM, Wusthoff M (2009) Service specification in cloud environments based on extensions to open standards. In: Fourth International Conference on COMmunication System softWAre and middlewaRE (COMSWARE 2009)
14. Ganek A, Corbi T (2003) The dawning of the autonomic computing era. IBM Systems Journal 42(1):5–18
15. IBM (2007) IBM blue cloud. [Online] http://www-03.ibm.com/press/us/en/pressrelease/22613.wss
16. Mamatas L, Clayman S, Charalambides M, Galis A, Pavlou G (2010) Towards an information management overlay for the future internet. In: IEEE/IFIP NOMS 2010
17. Massie ML, Chun BN, Culler DE (2003) The ganglia distributed monitoring system: Design, implementation and experience. Parallel Computing 30:2004
18. Microsoft (2008) Windows Azure Platform. [Online] http://www.microsoft.com/windowsazure/
19. Nagios (1999) Nagios. [Online] http://www.nagios.org/
20. Newman H, Legrand I, Galvez P, Voicu R, Cirstoiu C (2003) MonALISA : A distributed monitoring service architecture. In: Proceedings of CHEP03, La Jolla, California
21. OASIS (2007) Web Service Business Process Execution Language Version 2.0 Specification. OASIS standard
22. Object Management Group (2006) Meta Object Facility Core Specification 2.0, OMG Document, formal/2006-01-01
23. Object Management Group (2006) Object Constraint Language (OCL) 2.0, OMG Document, formal/2006-05-01
24. OpenNebula (2010) OpenNebula: The Open Source Toolkit for Cloud Computing. [Online] http://www.opennebula.org
25. Rochwerger B, Breitgand D, Levy E, Galis A, Nagin K, Llorente L, Montero R, Wolfsthal Y, Elmroth E, Cáceres J, Ben-Yehuda M, Emmerich W, Gálan F (2009) The RESERVOIR Model and Architecture for Open Federated Cloud Computing. IBM Systems Journal Special Edition on Internet Scale Data Centers 53(4)
26. Rochwerger B, Galis A, Levy E, Cáceres J, Breitgand D, Wolfsthal Y, Llorente I, Wusthoff M, Montero R, Elmroth E (2009) RESERVOIR: Management Technologies and Requirements for Next Generation Service Oriented Infrastructures. In: The 11th IFIP/IEEE International Symposium on Integrated Management,(New York, USA), pp 1–5
27. SAP (2003) SAP Enterprise Resource Planning. [Online] http://www.sap.com/solutions/business-suite/erp/index.epx

28. Skene J, Emmerich W (2005) Engineering runtime requirements-monitoring systems using MDA technologies. Lecture notes in computer science 3705:319
29. Skene J, Lamanna DD, Emmerich W (2004) Precise service level agreements. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, pp 179–188
30. Srinivasan R (1995) XDR: eXternal Data Representation standard
31. Sugerman J, Venkitachalam G, Lim BH (2001) Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In: Proc. of the 2001 USENIX Annual Technical Conference, Usenix, Boston, Mass
32. Tatemura J (2006) CDDLM Configuration Description Language Specification 1.0. Tech. rep., Open Grid Forum
33. Thain D, Tannenbaum T, Livny M (2002) Condor and the Grid. In: Berman F, Fox G, Hey T (eds) Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc.
34. UCL (2008) UCL MDA Tools. [Online] http://uclmda.sourceforge.net/
35. Vaquero LM, Rodero-Merino L, Cáceres J, Lindner M (2009) A break in the clouds: towards a cloud definition. SIGCOMM Comput Commun Rev 39(1):50–55, DOI http://doi.acm.org/10.1145/1496091.1496100
36. Warmer J, Kleppe A (2003) The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA