2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications

Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
E-mail: {e.copil, d.moldovan, truong, dustdar}@dsg.tuwien.ac.at

***Abstract*—Elasticity in cloud computing is a complex problem, regarding not only resource elasticity but also quality and cost elasticity, and most importantly, the relations among the three. Therefore, existing support for controlling elasticity in complex applications, focusing solely on resource scaling, is not adequate. In this paper we present SYBL – a novel language for controlling elasticity in cloud applications – and its runtime system. SYBL allows specifying in detail elasticity monitoring, constraints, and strategies at different levels of cloud applications, including the whole application, application component, and within application component code. Based on simple SYBL elasticity directives, our runtime system will perform complex elasticity controls for the client, by leveraging underlying cloud monitoring and resource management APIs. We also present a prototype implementation and experiments illustrating how SYBL can be used in real-world scenarios.**

***Keywords*- elasticity, cloud computing, elasticity specification**

I. Introduction

The cloud computing paradigm has increased drastically the dynamism of resource provisioning, enabling a rapid development of elastic cloud applications. Although elasticity plays an important role in nowadays cloud computing infrastructures, it is typically regarded only from the resources elasticity point of view [1]–[3]. For instance, Amazon [4] provides the autoscale service, which allows the definition of policies for automatically scaling the application. ElasticHosts [5] and CloudSigma [6] are two of the providers which have, as selling point for their infrastructures, assuring elasticity. However, elasticity is a multi-dimensional view, such as one can scale up/down the quality or the cost of the applications, rather than just the resources, as shown in recent studies [7]. Unfortunately, little focus has been devoted to defining elasticity and controlling it from a multi-perspective view and to allowing customers to specify complex and changing elasticity requirements throughout their application's execution on a cloud infrastructure.

In our work, we consider elasticity a multi-dimensional issue, defined by the relation between elastic properties classified into three main dimensions namely cost, quality and resource. In this view, the user should be able to specify constraints and relations for the aforementioned dimensions. To this end, our approach is to develop programming elasticity directives. Programming directives are well-known, simple-to-use, and efficient means of controlling work distribution and communication at runtime. We believe that similar directives but for controlling elasticity can also hide the complexity of writing and executing elasticity strategies from the user.

In this paper, we contribute the detailed design and implementation of SYBL (Simple Yet Beautiful Language) – a novel language for controlling elasticity in cloud applications – and its runtime system for controlling elasticity in cloud applications. SYBL can be used for three specification levels: application level, component level and programming level. The elasticity specification at application level gives a high level description of user's preferences regarding the application to be deployed on the cloud infrastructure. With a higher level of granularity, component level elasticity requirements refer to the components constituting the application, while programming level elasticity requirements specification deals with code-level elasticity requirements description. The component level elasticity requirements have been tackled by several papers so far [3], [8], [9], but without a clear focus on cloud application elasticity. To the author's knowledge, programming level elasticity requirements specification has not been approached before, mostly due to the complexity of elasticity in cloud computing.

SYBL can be used by application developers, software providers, IaaS or PaaS end-users and cloud providers. This enables a multitude of applications in which SYBL can be used to control the elasticity. For example, a cloud customer can try to achieve trade-offs on cost and quality, or just try to minimize the price the customer has to pay. On the other side, SYBL can be used by cloud providers to specify generic elasticity strategies for the applications hosted on their infrastructures. Therefore, SYBL enables many types of users to specify the elasticity behavior of an application without having to use complex cloud APIs or monitoring tools.

The rest of this paper is organized as follows: Section 2 presents our view on elasticity requirements, Section 3 gives a semantic and syntactic description of SYBL, Section 4 describes the experimental work achieved using SYBL. Section 5 presents related work, while Section 6 presents the concluding remarks and future work.

This work was supported by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790). We thank Yike Guo and Benjamin Satzger for their fruitful discussion.

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.42

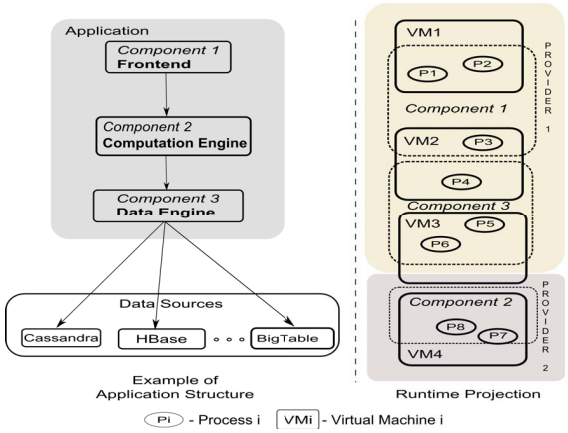


Figure 1: Illustrative application structure and its deployment

II. Elasticity requirements

Elasticity is usually referred-to just in terms of resources, without considering implications from the point of view of other properties an application may have [1], [3], [9]. When talking about elasticity we refer to the capacity of an application to change or to be changed according to the context in which it resides. Elasticity targets not just resources and their capacity to scale, but also their relations with the different types of costs and quality, and the capacity of an application to oscillate between different states (each state being described by resources, cost, quality and their attributes) for obtaining best quality at best price possible.

For describing what types of elasticity controls could be required and the complexity of elasticity requirements, we examine elasticity requirements from different types of clients. Let us consider a generic application model of which the structure is given in Figure 1. The application has different components, including a front-end, a computation engine, and a data engine. Such an application is quite popular, e.g., for large-scale data analytics, in which the front-end accepts requests from clients, the computation engine can be implemented as parallel applications, and the data engine may have an extensive, varying amount of calls to different data sources from HBase [10] to Cassandra [11] or user-defined repositories. As a result, each component consists of multiple processes, which at runtime may need to scale depending on user requirements which are parametrized by the number of users, the difficulty of the computations and the type of processes. Each process of the component has certain elasticity requirements at deployment. Depending on the user-defined elasticity specifications, the processes can be scaled individually either by creating more process instances in the same or different virtual machines, or by allocating more or less computing resources.

The elasticity requirements are demands formulated by users that they consider necessary and that contain necessary conditions for the application to be an elastic one. Elasticity can be a subjective notion and depends on the type of the user and the granularity at which he/she wants to specify the application's elasticity. On one hand, the purpose of elasticity requirements varies from controlling costs to achieving higher quality or even specifying demands on the relation between cost, resources and quality, for example:

- *Cost-related elasticity requirements*: A cloud customer may specify that when the total cost is higher than 800 Euro, there should be a scale-in action for keeping costs in acceptable limits.
- *Quality-related elasticity requirements*: A cloud user may need to monitor different quality parameters, which should be in acceptable limits. For instance, a software provider can specify constraints on the response time depending on the number of users currently accessing the provided software. A developer could specify that the result from a data analytics algorithm must reach a certain data accuracy under a cost constraint without caring how many resources should be used for executing the code of that algorithm.
- *Elasticity requirements on the relation between cost and quality*: A cloud provider could specify its pricing schema or price computation policies, for example that when availability is higher than 99% the cost should increase by 10%.

On the other hand, the user needs different granularities at which he/she can specify elasticity requirements, therefore, having specifications enforced at different levels, as opposed to usual approaches in resource allocation and reallocation such as to control the resource only at the whole application or component level [3], [9], [12]. To this end, the following elasticity controls at different levels should be supported:

- *Application level elasticity requirements*: elasticity requirements can be applied on the overall availability of the whole application, imposing aggregating data about the different components of the application and the communication between them. The cost for the application at a defined level, refers to the cost of components usage, communication between components and storage.
- *Component level elasticity requirements*: the user could specify different requirements based on the component type, i.e. the nature of requirements for the computation engine that are different from the requirements for the front-end component. The user can control the cost associated with a component, which means aggregating the cost of processes, storage and communication associated to the component but residing on different virtual machines.
- *Programming level elasticity requirements*: the user could need to specify that for some specific code the CPU size and memory should be really high, when the cost is high. In this case, when specifying requirements

about the cost, the developer refers to the cost for running the specific portion of code.

To support the above-mentioned requirements, we characterize elasticity properties into a "resources-cost-quality" representation in Figure 2 where each axis contains sub-dimensions of cloud application requirements specification. The user should be given the opportunity of specifying the application's behavior, most specifically in what direction it should automatically scale in different cases in the space defined by the three axes (resource, cost and quality). Many properties from each of the elasticity dimensions are strongly interdependent, an elasticity property belonging to one axis being a multi-dimensional function of properties belonging to the other two axes.

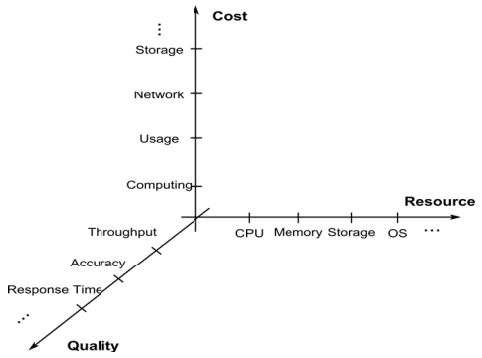


Figure 2: Common dimensions for application elasticity

We, therefore, need a novel approach towards elasticity control specifications. Our approach is to use programming directives that can be used to monitor and specify different elasticity constraints and strategies. Programming directives for controlling elasticity should be easy to use. However, they should be generic and extensible.

III. SYBL Syntax and semantics

A. SYBL overview

The initial idea of using elasticity directives for cloud computing is first described in [13], targeting programming level directives in elastic computing. The directive-based elasticity specification can substantially reduce the overhead by delegating the control to underlying elasticity middleware. SYBL is designed for that purpose in order to meet requirements mentioned in Section II. Before describing in detail how we design and implement directives for elasticity controls, we will overview the concepts of elasticity programming directives in this section.

SYBL enables elastic specifications at different granularities, depending on the user's demands and perspective: application, component and programming (see Figure 1). As opposed to usual mechanisms of reallocations, where rules (mostly SLAs) are specified at a per-application level, SYBL provides greater elasticity granularity. At the highest level, the application level, certain global application characteristics can be described. At the component level we can express a lower level description for black-box components or elasticity requirements focusing at a level lower than application but higher than code, while the programming level enables the user to specify elasticity requirements at code level. One may argue on the need of application level specifications, when the user can specify broadly all the requirements at programming and at component level. This application level meets cloud users who may want to deploy their application as a black-box, or cloud providers who receive the application as a black box without having any permissions to access it. By providing these finer-grained specification levels, we enable the user to decide where elasticity specifications could or should be placed, considering application's elasticity requirements.

The SYBL language, being a directive-based one, is easy to understand and use, while aiming of high expressiveness. The elasticity specification described in the previous section would be a strategy "`averageCost > 300 Euro: scaleIn`". This would simplify the complexity of using the cloud API and multiple monitoring tools calls to implement elasticity controls as well as enable the multi-dimensional elasticity controls, which are not well supported currently. Using programming level elasticity directives, the user can focus on defining the strategy and on the appropriate measures he/she should take.

SYBL empowers the user to design elasticity specifications which reference other elasticity specifications. After defining a complex constraint, the user can simply specify strategies about its violation or fulfillment, without having to re-describe it as a condition for the new strategies. It also enables a hierarchical description of elasticity specifications, for cases of overlapping constraints or strategies or unclear elasticity descriptions.

B. Language constructs

1) Predefined functions: SYBL includes several predefined functions which regard two different kinds of information: information on the current environment and information on the elasticity specifications. The environment comprises different types of static and dynamic cloud information. When we refer to the environment, we have to consider the level at which each of these predefined functions or variables appear. As described before, the information differs in the different levels of elasticity requirements, and therefore the environment which comprises all the information at a given moment in time will also differ based on these levels. Accessing information on the current environment enables the user to be aware of capabilities of the underlying cloud computing infrastructure and the current application state.

SYBL contains several predefined functions like $GetEnv$ which can be used to obtain the current environment (taking also into account the level from which it is called),

Function	Description
GetEnv	Current cloud infrastructure environment
Violated	Checks whether the constraint sent as parameter is violated
Enabled	Checks whether an elasticity specification is enabled or not
Priority	Returns the priority of an elasticity specification

Table I: Example of predefined functions

$Violated/Fulfilled$ returns true or false, depending on whether or not the constraint received as a parameter is fulfilled, the $Priority$ function associates to each elasticity specification a priority (see Table I). The environment variables refer to general information like the currently considered compute bid, the cost spent, etc. (see Table II). The environment variables depend on the level at which they appear in elasticity specifications. The environment variables have at their source more complex function calls which are used frequently. For example, `optimal_cloud_provider` variable hides a call to `GetEnv().findOptimalCloudProvider()`.

Environment variable	Description
optimal_cloud_provider	The cloud provider that the decision components finds to be best suited
compute_bid	The current bid for the current cloud provider
total_cost	The cost - depends on the level at which variables are being referenced

Table II: Examples of predefined environment variables

Equations 1 and 2 formally describe the set of variable and functions predefined in SYBL, which will be later used for the description of monitoring, constraints, and strategies.

$$DefFunctions := \{GetEnv, Balance, Violated, Enabled, Priority\} \quad (1)$$

$$EnvVariables := \{compute\_bid,\ total\_cost, optimal\_cloud\_provider\} \quad (2)$$

2) Monitoring directives: The monitoring directives start with the `MONITORING` keyword and assign to a new variable some types of cloud information to be monitored (see Equation 3). The monitoring variable is assigned existing information or formulas constructed for combining several types of cloud information.

$$M_i := \texttt{MONITORING}\ varName = x_j \mid \texttt{MONITORING}\ varName = formula(x_1 ... x_n) \ where \ x_j \in c, c \in\ ApplicationDescriptionInfo \quad (3)$$

3) Constraints directives: A constraint describes the limits in which the current application's description can oscillate. As shown in Equation 4 a constraint directive starts with the keyword `CONSTRAINT`, and uses mathematical signs ($<, >, >=, <=, !=, ==$) for reflecting which values are admissible. Constraints can be established on a simple type of cloud information or on a complex type of cloud information determined by formulas ($formula_i$ or $formula_j$).

$$C_i := \texttt{CONSTRAINT}\ p\ \in\ formula_i(x)\ rel\ formula_j(y) \ where \ x, y\ \in\ ApplicationDescriptionInfo \ rel\ \in\ \{\leq, \geq, \neq, =\} \quad (4)$$

4) Strategies directives: A strategy describes a recipe to be followed in case the triggering condition becomes true. A strategy starts with the keyword `STRATEGY` and usually has the form Condition:Action or `WAIT` Condition (see Equation 5). The first pattern triggers the execution control action (e.g., deploy, migrate, delete, scale) specified in case the condition is true, while the second pattern waits for the condition to be true. The condition can also refer to fulfillment or violation of constraints, and actions to be taken in those cases.

$$S_i := \texttt{STRATEGY CASE}\ [Condition : Action] \mid \texttt{WAIT}\ Condition \mid \texttt{STOP} \mid \texttt{RESUME} \mid \texttt{EXECUTE}\ strategyName\ parameter_{1...n} \ where \ Condition : DefFunctions \rightarrow \{true, false\} \quad (5)$$

C. SYBL Runtime

The SYBL runtime takes elasticity specifications and carries out elasticity controls at runtime. In the SYBL runtime, elasticity requirements expressed through SYBL will be interpreted, processed by the *Control Service* and then enforced by the use of cloud APIs. The *Control Service* is the central part of the runtime system, being the component which handles the actual coordination between the specified state of the application from user's perspective, and the current application elasticity state.

The deployment of SYBL runtime system from Figure 3 mainly concerns the SYBL programming directives, which are more difficult to be enforced since they need to be caught with higher precision, but also applies to component and application level directives. The *SYBL Local Interpreter* is instantiated within each process containing SYBL directives. The *SYBL Local Interpreter* catches the SYBL directives, interprets them and forwards the requests to the *Local Service*. The latter is a part of the *Control Service*, is deployed and resides on the VM and enforces the different elasticity requirements coming from processes of the same application. The main *Control Service* communicates with *Local Services* and correlates received information for enabling the enforcement of requirements at component

level and application level, even when the component or application processes do not reside on the same VM. The *Control Service* also communicates with cloud APIs and monitoring tools for having an up-to-date knowledge about the application elasticity state.

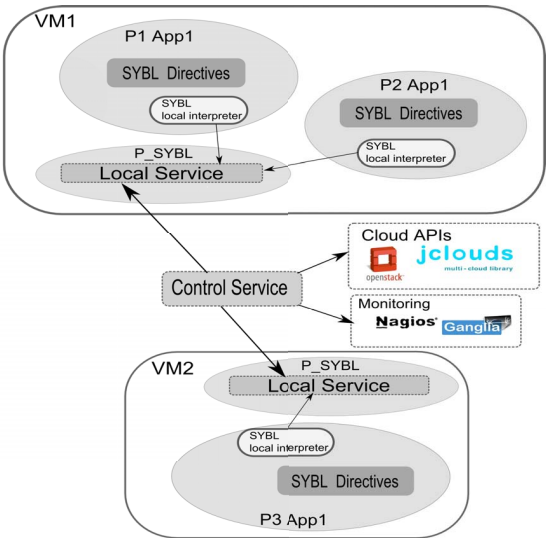


Figure 3: SYBL based control at runtime

The *Control Service* uses monitoring tools and cloud APIs for providing the necessary functionality. The multitude of cloud APIs and monitoring tools that elasticity requirements depend on can seem overwhelming. The need for more than one tool with which the SYBL runtime interacts is obvious, since a single tool could not provide a complete description of the cloud infrastructure capabilities and generic mechanisms of monitoring and interacting with the cloud infrastructure. For instance, Nagios [14] and Ganglia [15] are many times used together due to the fact that they compensate each-other in the information provided and the mechanism of collecting data.

These tools are in a continuous change and this is why the language should not be bound to a specific cloud API. However, the *Control Service* should be able to interface to the tools specific to different cloud providers (e.g. via plug-in mechanisms). SYBL is designed to be extensible: that is, to have the capacity of enveloping new concepts without much difficulty. SYBL users can easily add metrics and concepts they need to focus on, the extensibility property being one of the strong points of SYBL language.

D. Examples of elasticity controls

The following SYBL examples do not refer to a specific language implementation, the intention being simply to show that short and simple SYBL elasticity requirements hide the actual complex implementation and enforcement layers.

Listing 1 shows how a cloud provider can actualize the price perceived for the current application depending on application's availability so far. The strategy $Str1$ specifies the price for the case in which the availability is greater than 98%. The strategy $Str2$ overrides the previous specification, stating that an availability larger than 99% should set the price at 300 Euro.

Listing 1: SYBL elasticity requirements - cloud provider

```
#SYBL.ApplicationLevel
Str1: STRATEGY CASE availability>98% setPrice(100)
Str2: STRATEGY CASE availability>99% setPrice(300)
Priority(Str1)<Priority(Str2)
```

Listing 2 shows possible elasticity requirements from the application developer side. Constraint $Component2.Cons5$ specifies that the CPU usage in the computation engine should be less than 80% for avoiding performance degradation due to high CPU load. Constraints $Cons3$ and $Cons4$ overlap in the sense that both refer to the costs, but at different levels: $Cons4$ specifies that the cost of hosting the data engine should be less than 600 Euro, while $Cons3$ refers to the cost of hosting the entire application. The programming level elasticity requirement encompasses the sequence of code, which for this case can be a data analysis algorithm, setting constraints about data accuracy and costs, without having specific resource-related requirements.

Listing 2: SYBL elasticity requirements - developer

```
#SYBL.ApplicationLevel
Mon1: MONITORING rt = Quality.responseTime
Cons1: CONSTRAINT rt < 2 ms. when nbOfUsers < 1000
Cons2: CONSTRAINT rt < 4 ms. when nbOfUsers <
    10000
Cons3: CONSTRAINT totalCost < 800 Euro
Str1: STRATEGY CASE Violated(Cons1) OR Violated(
    Cons2): ScaleOut
Priority(Cons1)=3, Priority(Cons2)=5
#SYBL.ComponentLevel
ComponentID = Component3; ComponentName= Data
    Engine
Cons4: CONSTRAINT totalCost < 600 Euro
#SYBL.ComponentLevel
ComponentID = Component2 ComponentName= Computing
    Engine
Cons5: CONSTRAINT cpuUsage < 80%
#SYBL.ProgrammingLevel
Cons6: CONSTRAINT dataAccuracy>90% AND cost<400
```

These examples show the ease of specifying elasticity requirements with SYBL and the power that lays underneath the simple description. All these elasticity requirements are enforced with no effort from the user side, and transformed into calls to different APIs and tools that help at enforcing these elasticity requirements.

For cases where users (software providers and cloud providers) have a black box application/component for which they need to specify elasticity requirements, elasticity specifications can be annotated inside the XML-based descriptions of the application/component (e.g. OVF). Listing 3 shows how a specific elasticity specification can be integrated into the already existing sections of the OVF format.

The resource ranges can be integrated into the already existing OVF structure, e.g. the specification bellow states that the minimum amount for memory (OFV resource type with value 4) should be greater than 384 MB. The elasticity requirements for quality and cost are added as an XML structure in the additional section of OVF.

Listing 3: Example of elasticity constraints in OVF

```
<VirtualHardwareSection><Item ovf:bound="min">
    <rasd:InstanceID>0</rasd:InstanceID>
    <rasd:Reservation>384</rasd:Reservation>
    <rasd:ResourceType>4</rasd:ResourceType>
</Item> </VirtualHardwareSection>
```

IV. Experiments

We have developed a prototype of SYBL[1] containing a partial implementation of its runtime system. We currently support the SYBL elasticity specifications for processes residing inside the same VM, the implementation of the *Control Service* (Figure 3) being in our focus as future work. We support SYBL elasticity specifications as Java annotations processed at runtime by AspectJ or as SYBL enriched XML descriptions, which are interpreted and then enforced through the *Local Service*. We tested the current prototype on our local cloud running OpenStack [16]. In the current implementation, Ganglia [15] provides information on computing resources allocated to virtual machines and their usage, the number of packets sent and received by each virtual machine, etc. JClouds [17] is used mainly for scaling the current instances and for controlling them.

A. Experimental application

The structure of our experimental application is shown in Figure 4 and described in Section II. In our experiments we focus on illustrating the feasibility of SYBL specifications.

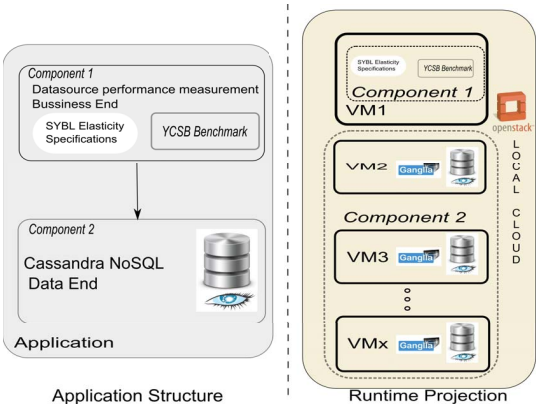


Figure 4: Application structure used for experiment

For this experimental application we focus on the data engine and its ability to be dynamically controlled by the SYBL user. As data engine we use Cassandra - a NoSQL distributed database system, while at the business side we use the YCSB [18] benchmark for simulating different types of workload. The application is mainly composed of a virtual machine hosting the business end of the application and a cluster of virtual machines hosting Cassandra nodes. The workloads used represent a combination of read, write, scan and update operations, actually characterizing real-life applications. For example, a read-oriented workload can be related to an application focused on photo-tagging, where the largest part of the operations is reading tags. The user of this application may have the following elasticity requirements:

- *Application level elasticity requirement*: the user may need for the overall cost of his application being hosted for this experiment to be less than 200 Euro
- *Component level elasticity requirement*: the user could specify strategies for the case in which the average number of IOs is too small, this way generating one more component business component and therefore more workload
- *Programming level elasticity requirement*: the user may want for his application data-end to scale dynamically when running the workload, for keeping the CPU usage in admissible values.

B. SYBL annotations-based elasticity requirements

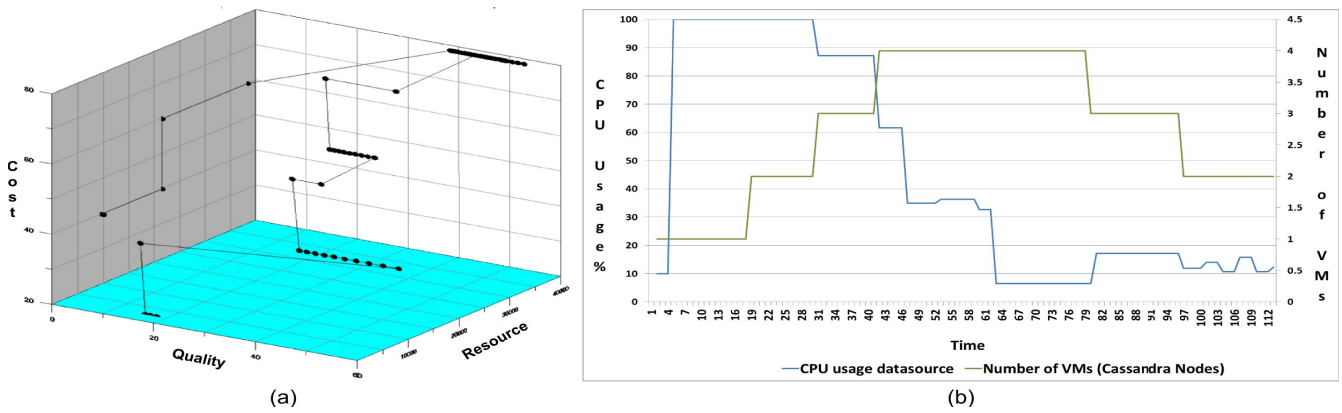
The SYBL annotation (see Listing 4) states at component level that in case the average number of IOs is less than 50 application should scale out (strategy $St1$) for introducing a higher workload. Strategies $St2$ and $St3$ of the component level annotation enforce the two constraints specified, for keeping resource utilization in acceptable ranges.

Listing 4: Component-level SYBL annotation

```
@SYBL_ComponentContext(constraints=
"Co1:CONSTRAINT memory.usage<80% AND cpu.usage
    <80%;
Co2:CONSTRAINT memory.usage>20% AND cpu.usage>20%"
,strategies=
"St1:STRATEGY CASE IO.averageNb<50:ScaleOut;
St2:STRATEGY CASE Violated(Co1): ScaleOut;
St3:STRATEGY CASE Violated(Co2): ScaleIn")
```

The programming level SYBL specification from Listing 5 annotates a method executing workload that generates database operations. The user specifies an elasticity requirement that the data source should automatically scale and keep the CPU usage on the data engine at predefined levels. From the three specified constraints just two are always enabled. This is due to the higher priority of constraint $Co3$ next to the constraint $Co1$. The strategies refer to the constraints giving scaling advises with respect to the data source for the two mentioned cases. As a result of producing the workload referred by this method, the database will scale dynamically as we will see in the next subsection.

[1]Detailed prototype implementation and further experiments can be found at http://dsg.tuwien.ac.at/prototypes/SYBL/index.html.



(a) (b)

Figure 5: (a) Evolution of application in elasticity space (b) CPU usage correlation with the number of VMs used

Listing 5: Programming-level SYBL annotation

```
@SYBL_ProgrammingContext(type=AnnotType.DURING,
 constraints="Co1:CONSTRAINT cpuUsageData < 65;
  Co2:CONSTRAINT cpuUsageData > 30;
Co3:CONSTRAINT cpuUsageData < 85 WHEN cost > 70",
 monitoring ="Mo1:MONITORING cost = cost.instant;
  Mo2:MONITORING dataThroughput = throughput.
      datasource;
  Mo3:MONITORING cpuAllocatedData = cpu.size.
      datasource;
  Mo4:MONITORING cpuUsage = cpu.usage;
  Mo5:MONITORING cpuUsageData = cpu.usage.
      datasource",
 strategies=
 "St1:STRATEGY CASE Violated(Co2):
     scaleInDataSource;
  St2:STRATEGY CASE Enabled(Co1) AND Violated(Co1)
      : scaleOutDataSource;
  St3:STRATEGY CASE Enabled(Co3) AND Violated(Co3)
      : scaleOutDataSource;",
 priorities="Priority(Co3) > Priority(Co1)")
```

C. Results

Based on elasticity directives specified in Listing 5, the elasticity runtime behavior of the data engine is scaled on multiple instances hosting Cassandra nodes, considering the CPU usage and cost metrics. Figure 5a shows the elasticity evolution in terms of cost, quality and resources, each three-dimensional point being a three-dimensional state of the data source part of the application. The quality in this case refers to the throughput, the resources refer to the allocated CPU size while the cost is estimated based on the resources allocated. The chart shows how based on the SYBL elasticity annotations the application evolves in the elasticity space, adjusting the resources, cost and quality to the user's elasticity requirements. Figure 5b gives a view on the evolution of the application on the data engine in terms of number of instances and CPU used: with the increase of instance number, produced by the SYBL strategies, the CPU usage decreases.

This experiment reveals SYBL's strength, enabling users to describe the application elasticity requirements from the three perspectives: monitoring, strategies, and constraints. We have shown how SYBL and its runtime system can ensure through the business level elasticity specifications the elastic control of data sources, the user being able to specify his/her requirements at the desired granularity, and depending on the data that needs to be taken into consideration. The results show that the application elasticity has accorded with specified SYBL directives following accurately the desires of the SYBL users.

V. Related Work

Resource re-allocation and requirements specification have been a focus usually from the SLA fulfilment or scheduling and resource allocation perspectives. Macias et al. [19] provide an evaluation of possible SLA administration strategies, showing how cloud providers revenues change when optioning for different strategies. Fard et al. [20] approach static scheduling with a different view, defining a multi-objective optimization algorithm and demonstrating its usefulness on real-world applications. Han et al. describe in [12] an approach for fine-grained scaling at resource level in addition to the VM-level scaling, which uses a lightweight scaling algorithm for improving resource utilization while reducing cloud providers' costs. Our approach differs from these works in two main points: we support (i) multiple levels of elasticity controls using (ii) multiple elastic properties.

In [1] the authors present an attempt to tackle the problem of elasticity from the point of view of resource and elasticity in SaaS based clouds. The authors propose relating cost with quality: cost per performance metric (C/P) and cost per throughput (C/T). Li et al. [21] review the existent metrics for evaluating commercial cloud services from economics evaluation metrics to elasticity evaluation metrics setting the focus on the complexity of cloud computing environments. Sharma et al. [2] propose a framework for cost optimization, considering the fact that the resources, the cost and the quality obtained influence each other. Therefore, the idea of application elasticity in cloud as a complex

multi-dimensional problem is not new, and research effort goes into providing solutions or partial solutions. However, existing works have not developed flexible languages for controlling multi-dimensional elastic properties.

In [8] Galan et al. propose an extension of OVF for service specification in cloud environments describing resource as well as business rules and enforcing them through resource allocation/de-allocation. Chapman et al. [3] describe an elastic service definition in computational grids. Morán et al. [9] provide a rule-based approach for specifying application requirements. In contrast with these two approaches, our main focus is describing elasticity requirements and the different granularities at which they can be specified by developers, end-users or cloud providers.

The major difference between existing work and our approach is that our work tackles elasticity requirements from more than one perspective (resource, quality, cost) and at different levels of granularity, thus assigning the user the capacity of specifying when the application should scale throughout its execution and, most importantly how.

VI. Conclusions and Future Work

This paper presents SYBL and its runtime system for controlling elasticity requirements in cloud applications. SYBL enables a wide range of users, from developers to cloud customers and cloud providers, to specify the application's elasticity in a simple way, while the actual complex enforcement of elasticity remains transparent to the users. SYBL features, covering different elasticity constraints, strategies and monitoring directives, permit a wide range of flexible ways for controlling application's elasticity, while for cases where some needed metrics are not yet present in SYBL, the language can be easily extended to reflect the elastic properties of user's focus.

As future work, we intend to extend our implementation of the SYBL runtime stack and utilize it in a fine-grained elasticity module for various types of applications from gaming to data-intensive applications, addressing elasticity requirements issues which depend on application type. Furthermore, new elasticity directives for controlling data elasticity will be covered.

References

[1] P. Martin, A. Brown, W. Powley, and J. L. Vazquez-Poletti, "Autonomic management of elastic services in the cloud," in *Proceedings of the 2011 IEEE Symposium on Computers and Communications*, ser. ISCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 135–140.

[2] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, june 2011, pp. 559 –570.

[3] C. Chapman, W. Emmerich, F. G. Marquez, S. Clayman, and A. Galis, "Elastic service definition in computational clouds," Apr. 2010, pp. 327–334.

[4] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[5] Elastic Hosts. http://www.elastichosts.com/.

[6] Cloud Sigma. http://www.cloudsigma.com/.

[7] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of elastic processes," *Internet Computing, IEEE*, vol. 15, no. 5, pp. 66 –71, sept.-oct. 2011.

[8] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, and L. M. Vaquero, "Service specification in cloud environments based on extensions to open standards," in *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, ser. COMSWARE '09. New York, NY, USA: ACM, 2009, pp. 19:1–19:12.

[9] D. Morán, L. M. Vaquero, and F. Galn, "Elastically ruling the cloud: Specifying application's behavior in federated clouds," in *IEEE CLOUD'11*, 2011, pp. 89–96.

[10] HBase NoSQL Database. http://hbase.apache.org/.

[11] Cassandra NoSQL Database. http://cassandra.apache.org/.

[12] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 644–651.

[13] S. Dustdar, Y. Guo, R. Han, B. Satzger, and H.-L. Truong, "Programming directives for elastic computing," *Internet Computing, IEEE*, p. (to appear), 2012.

[14] Nagios. http://www.nagios.org/.

[15] Ganglia monitoring system. http://ganglia.sourceforge.net/.

[16] OpenStack EC2 API. http://api.openstack.org/.

[17] JClouds API. http://www.jclouds.org/.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[19] M. Maciandas, J. Fito and, and J. Guitart, "Rule-based sla management for revenue maximisation in cloud computing markets," in *Network and Service Management (CNSM), 2010 International Conference on*, oct. 2010, pp. 354 –357.

[20] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 300–309.

[21] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a catalogue of metrics for evaluating commercial cloud services," in *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, sept. 2012, pp. 164 –173.