Variability Modelling for Elastic Scaling in Cloud Computing

Mohamed Lamine Berkane[1], Lionel Seinturier[2] and Mahmoud Boufaida[1]
[1]*Computer Science Department, LIRE Laboratory, University Constantine 2, Algeria*
[2]*University of Lille and Inria, CRIStAL UMR CNRS 9189, France*

Keywords: Elasticity, Cloud Computing, Feature Model, Self-adaptive Systems, Variability.

Abstract: Elasticity is an increasingly important characteristic for cloud computing environments, in particular for those that are deployed in dynamically changing environments. The purpose is to let the systems react and adapt the workload on its current and additional (in an autonomic manner) hardware and software resources. In this paper, we propose an approach that allows the combination of variability and reusability for modelling elasticity. The used approach is based on self-adaptive systems and feature modelling into a single solution. We show the feasibility of the proposed model through Znn.com scenario.

1 INTRODUCTION

Cloud computing represents one of major innovation in Information Technology (IT). It describes a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Mell and Grance, 2011).

One of the important characteristics provided by cloud computing is elasticity. This concept permits a system to adapt the workload on its current and additional (in an autonomic manner) hardware and software resources (Herbst et al., 2013). The management of elasticity can be split in two methods: horizontal scaling and vertical one. The horizontal scaling permits to scale resources by changing the number of virtual machines (adding more virtual machines or devices to the computing platform). The vertical scaling allows adding more CPU, memory and disk depending on the application memory, storage and network bandwidth (Paraiso et al., 2014; Bahag and Madisetti, 2013; Marshall et al., 2010).

In this paper, we propose an approach that enables the modelling of elasticity in a modular way. This approach eases the modelling of elasticity by using an approach based on self-adaptive systems (IBM, 2006) and feature modelling (Czarnecki and Eisencker, 2000; Czarnecki et al., 2005; Greenfield et al., 2004). The self-adaptive systems provide mechanisms for modelling the structure and the behaviour of systems that support elasticity. In addition, the feature model provides a solution for describing and implementing the commonalities and variabilities of systems components.

The proposed approach supports the modelling of an elasticity system for cloud computing with three layers: *Adaptation loop*, *Architectural Model*, and *Elasticity System*. These layers are ordered hierarchically from very abstract to very concrete. One of the important challenges of our work consists in providing a solution for separating the reusable parts from the system specific ones.

The remainder of this paper is organized as follows. Section 2 presents the architecture for modelling elasticity system. In Section 3, we present the structural variability of modelling and Section 4 the behavioural one. Section 5 uses a case study to demonstrate the feasibility of our approach with the Znn.com application, and Section 6 implements and evaluates this application. In Section 7, we present some related work, and finally Section 8 concludes and discusses some future work.

2 A MULTI-LAYER BASED ARCHITECTURE FOR MODELLING ELASTICITY SYSTEMS

In this section, we present our architecture that enables modelling elasticity in a modular way. The

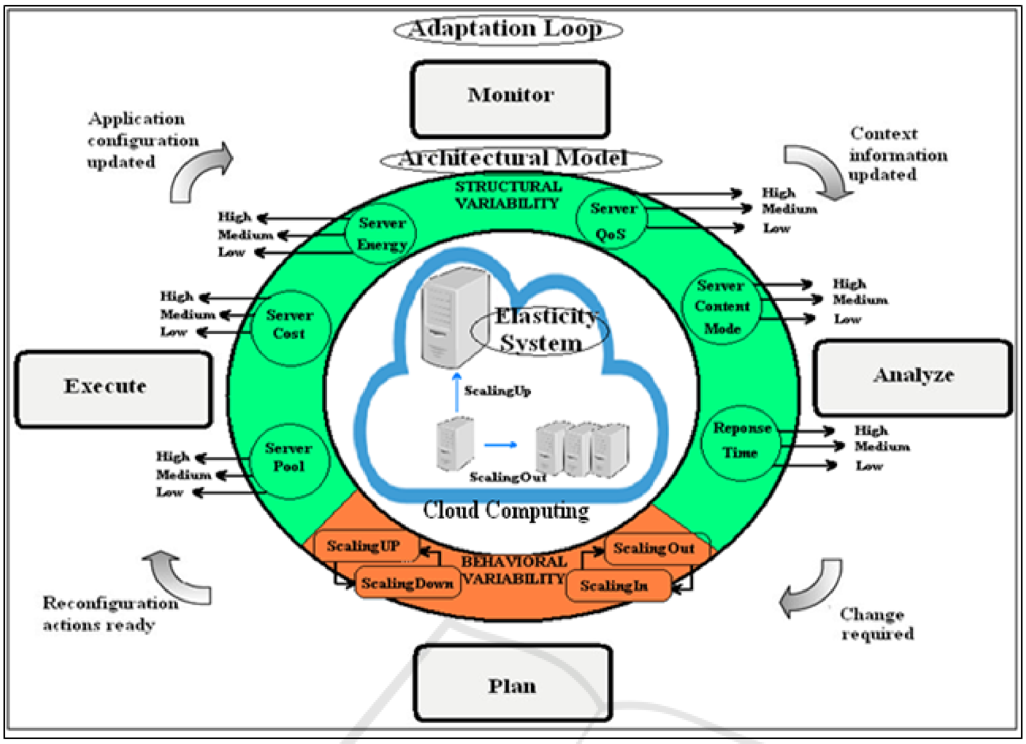

Figure 1: Overview of the proposed architecture.

proposed architecture defines three main layers: *Adaptation loop*, *Architectural Model*, and *Elasticity System* (Figure 1).

The *Adaptation loop* layer allows modelling the skeleton for the overall structure of the system. The *Architectural Model* layer defines the main functions and properties of the elasticity system in structural variability and behavioural one. The *Elasticity System* layer is equipped with a set of computing entities, such as sensors, which collect the information and actuators that change the state of the environment of cloud computing. To model elasticity, we use a reference model for autonomic control loops with four activities: *Monitor*, *Analyze*, *Plan*, and *Execute* (IBM, 2006).

These activities can be seen as components, which communicate (*Knowledge* component) to adapt elasticity's behaviour in response to change requirements and environmental conditions. In this layer, the *Analyze* component interprets the different data provided by the *Monitor* component (sensors). In addition, the *Analyze* compares the updated values found from the sensors with specific threshold values. Each threshold contains one or more boundary values. These values are used to represent boundaries between normal and abnormal behaviours. Once the *Analyze* indicates a situation when an adaptation might be needed, it creates a trigger to the *Plan* component. This last component selects the specific reconfigurations and show how the reconfigurations can be executed at run time by *Execute* component.

In the next section, we present the *Architecture Model* in two phases: structural variability and behavioural one.

3 STRUCTURAL VARIABILITY

This phase provides the structure of the system. It refines the application components by using feature modelling.

3.1 Extending Feature Model for Structural Variability

The *Feature Model* can be used to build the architectural model. It contains a set of *Features*, and the relationships between the *Features* like: implies and exclude (Czarnecki and Eisencker, 2000; Czarnecki et al., 2005; Greenfield et al., 2004). In our work, we define a new concept called *Feature Level*. It can represent the structure of system at different level from abstract to concrete. This new concept can be used to represent the autonomic control loop, properties, physical components and software ones of elasticity systems. Our new feature meta-model is shown in Figure 2.

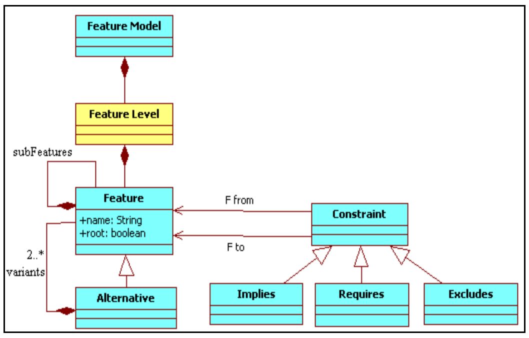

Figure 2: Feature Meta-model for structural variability.

3.2 Feature Model for Elasticity System

In the elasticity system, we define four kinds of layers: autonomic control loop, property, software component and physical one. These layers are specified by feature level (defined in feature meta-model). In the first layer, we represent the common architecture with *Monitor*, *Analyze*, and *Execute* component (IBM, 2006) as mandatory features (child features are required). In the other layers, we define the different functions related to the application as OR features (at least one of the child features must be selected). These functions represent the properties, the software components, and the physical ones related to model elasticity systems (Figure 3).

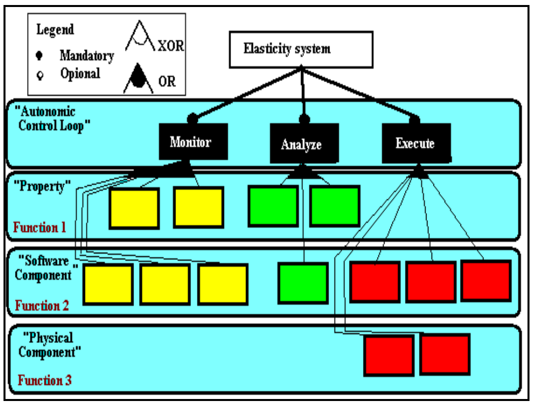

Figure 3: Structural variability.

The Plan component represents the different rules of elasticity system. It will be represented in the behavioural variability section.

4 BEHAVIOURAL VARIABILITY

In this phase, we specify the behaviour of the elasticity system. For this, we combine the feature model with a state transition diagram into a single solution.

4.1 Feature Model with State Transition Diagram

In this new model (state-transition part), each state is composed of the sets of components (software component and physical one). In addition (feature-model part), these components are associated to a set of configurations. The transitions between states represent the rules. These rules represent the different elasticity actions: *scaling up*, *scaling out*, *scaling in*, and *scaling down*. The variability is expressed by the different configurations and actions defined by the components (Figure 4).

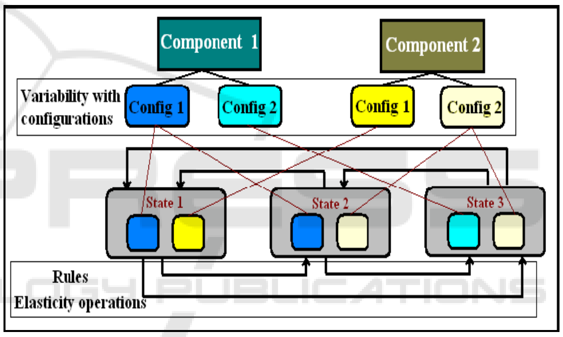

Figure 4: Behavioural variability.

4.2 Adaptation Behavioural of Elasticity

In this section, we propose an algorithm that considers the different scenarios of elasticity adaptation. This algorithm represents the different rules of elasticity system in *Plan* component.

We consider a system with a set of servers (**S**) and computing capacities (**C**). We define three parameters: **MaxS**, **MaxC** and **Q**. These three parameters represent respectively, the maximum number of servers in the pool, the maximum number of computing capacities, and the quality of service of the system (like response time) (see lines 1-5 in Listing 1). In the elasticity adaptation, we define four situations: The first situation determines the number of servers in the pool and the computing capacities under maximum (see lines 8-12 in Listing 1). In the second situation, we consider that the number of servers is in maximum and computing capacities under maximum (see lines 14-18 in Listing 1). The third situation is the reverse of the second one: the number of servers under maximum and computing capacities is in maximum (see lines 20-24 in Listing 1). In the last situation, we consider that the number of servers and the computing capacities are in maximum (see lines 26-30 in Listing 1).

We define four functions: **ScalingOut, ScalingUp, ScalingIn** and **ScalingDown** (see lines 35-56 in Listing 1).
ScalingOut: increment the number of servers. **ScalingUp:** increment the number of computing capacities. **ScalingIn**: decrement the number of servers. **ScalingDown:** decrement the number of computing capacities.

```
1 S: #Server
2 C: #ComputingCapacity (Memory)
3 MaxS:#max number of servers in the pool
4 MaxC: #max number of computing capacities
5 Q: QoS {Low, Medium & High}
6
7 Begin
8 If (S<MaxS & C<MaxC) then
9   If (Q=”Medium” | Q=”High”)
10   ScalingUp(C);
11  ElseIf (Q=”Low” )
12    ScalingOut(S);
13
14 If (S=MaxS & C<MaxC ) then
15  If (Q=”Low” | Q=”Medium”)
16     ScalingUp(C);
17    ElseIf (Q=”High”)
18     ScalingIn(S);
19
20 If (S<MaxS & C=MaxC)
21  If(Q=”Low”)
22     ScalingOut(S);
23   ElseIf(Q=”Medium” | Q=”High”)
24     ScalingDown(C);
25
26 If (S=MaxS & C=MaxC)
27  If (Q=”High”)
28   ScalingIn(S);
29  ElseIf(Q=”Low” | Q=”Medium” )
30    ScalingDown(C);
31 End
32
33 ScalingOut(S:Server):Server
34 {S<=S+1
35 Return(S)}
36
37 ScalingUp(C: ComputingCapacity):
ComputingCapacity
38 {C<=C+1
39 Return(C)}
40
41 ScalingIn(S:Server):Server
42 {S<=S-1
43 Return(S)}
44
45 ScalingDown(C: ComputingCapacity):
ComputingCapacity
46 {C<=C-1
47 Return(C)}
```

Listing 1: Adaptation behavioural.

5 CASE STUDY ‘Znn.com’

In this section, a case study is used to demonstrate the feasibility of our approach. We have selected a Znn.com scenario. Znn.com is an example that motivates the need for dynamic adaptation of elasticity system for cloud computing. It is one of the exemplar case studies proposed by the Software Engineering for Adaptive and Self-Managing Systems (SEAMS) research community. It is a web-based N-tier client-server system that provides news content to its customers. The main business objectives of Znn.com are (Cheng et al., 2009):

- News content provision in a reasonable response time
- Minimization of operating server costs
- High quality content presentation

5.1 Structural Variability

In the structural phase, we have defined four functions of Znn.com application (Figure 5): Response time (represent a property), Server pool (represent a physical component), Server cost (represent a property) and Server content mode (represent a software component). The Response time function is represented by three states (*Analyze* component): low, medium, and high. This function can only be observed by sensors. In addition, the server pool function has two actions (Execute component): Increment server pool and decrement one.

The first action increments the number of servers in the pool size. The second action is responsible for decrementing the number of servers in the pool size. However, the server cost function determines if we can increase the pool size (if the budget is under or over). In this function, we define two states: under budget and over budget. The last function specifies the server content mode: graphics and text. With this function, we can switch from graphical to textual mode, or from textual to graphical mode. The Plan component represents the different rules that can be executed to adapt Znn.com application. In the next section, we show the behaviour of this component.

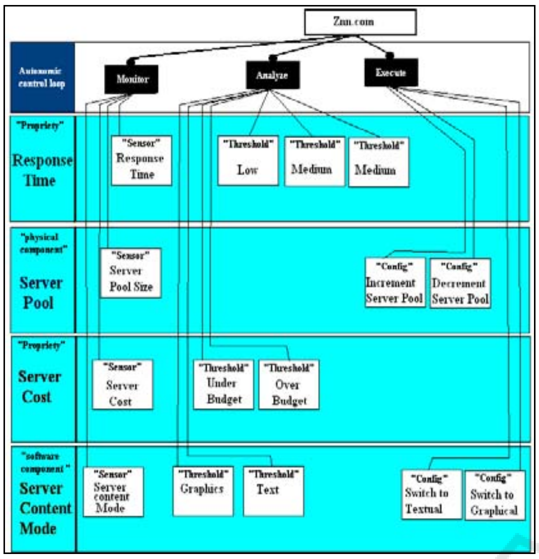
Znn.com
Autonomic control loop
Monitor
Analyze
Execute
"Propriety" Response Time
"Sensor" Response Time
"Threshold" Low
"Threshold" Medium
"Threshold" Medium
"physical component" Server Pool
"Sensor" Server Pool Size
"Config" Increment Server Pool
"Config" Decrement Server Pool
"Propriety" Server Cost
"Sensor" Server Cost
"Threshold" Under Budget
"Threshold" Over Budget
"software component" Server Content Mode
"Sensor" Server content Mode
"Threshold" Graphics
"Threshold" Text
"Config" Switch to Textual
"Config" Switch to Graphical

Figure 5: Structural representation of Znn.com application.

5.2 Behavioural Variability

In Znn.com application, we define four states. Each state is represented by three components: two components for two servers (physical components) and one component for server content mode (software component). For the server components, we can find two configurations: enable and disable. However, the server content mode component, we can find also two configurations: textual and graphical mode.

We represent three examples of scenarios (behaviours) of Znn.com (Cheng et al., 2009) with elasticity actions:

- Scenario 1: if the response time is high, Znn.com will increment the number of servers in the pool if the budget is not over (**scaling out**); otherwise, Znn.com will switch the servers to textual mode (**scaling down**);
- Scenario 2: if the response time is low, Znn.com will decrement its server pool size if it is near budget limit (**scaling in**); otherwise Znn.com will switch the servers to graphical mode if they are not already in that mode (**scaling down**);
- Scenario 3: if the response time is in the medium range, Znn.com will switch to graphical mode if it the mode is textual (**scaling down**), if the server pool size may either be incremented to decrease response time or decremented to reduce cost (**scaling out**).

In Figure 6, we present all the (behaviours) scenarios of Znn.com.

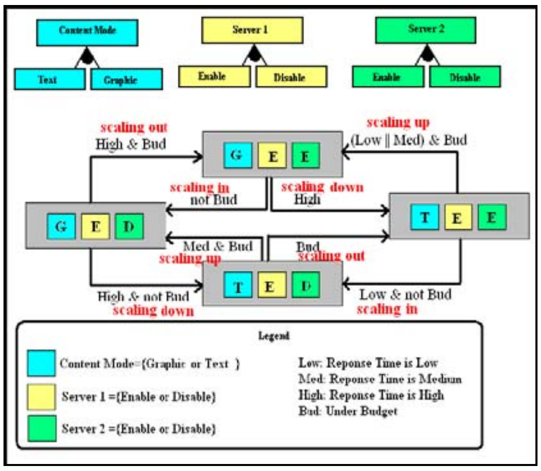

Figure 6: Behavioural representation of Znn.com application.

6 IMPLEMENTATION AND EVALUATION

We give some implementation aspects of elasticity in Znn.com.

6.1 Implementation

CaesarJ is selected as a language to implement Znn.com application. This language is an aspect-oriented language that supports reusability (Nunez and Gasiunas, 2015). It is an extension of Java and integrates the features model. This language defines five concepts to implement the application: *Collaboration Interfaces*, *Implementation*, *Binding*, *Weavlets* and *Weavlet Deployer* (Figure 7).

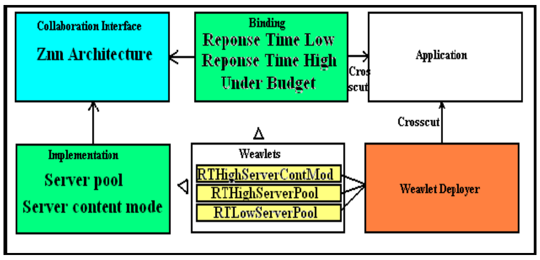

Figure 7: Znn.com application with CaesarJ language.

The modularity in CasearJ is materialized by *Collaboration Interfaces*. In these interfaces, one can separate the reusable parts from the application specific ones.

The Znn.com architecture is defined by the *Collaboration Interface* (Listing 2). In this Interface (represented by **cclass** concept), we define the different principal components of Znn.com application: Servers and Clients. In addition, this interface is split in a provided part and an expected one.

```
package znn;
abstract public cclass ZnnArch {
 abstract public cclass Serveur {
   abstract  public  void  AdaptAction
(Client cl);}
 abstract public cclass Client {
  abstract public float Request();} }
```

Listing 2: Znn.com Architecture with Collaboration Interface.

The provided part is implemented with reusable CaesarJ components "*Implementation*". This part implements the different actions related to Znn.com application (Listing 3). For server pool actions, we can find two actions: increment server pool and decrement one. However, the server content mode actions, we can find: textual and graphical mode.

```
package znn;
abstract public cclass ScreenMode
extends ZnnArch {
 abstract public cclass Serveur {
  private float resptime;
    public  void  AdaptAction  (Client
cl){
   float diff = cl.Resptime();
   resptime -= diff; }}
```

Listing 3: Implementation part of Znn.com.

The expected part is implemented with CaesarJ "*Binding*". It implements the variabilities of system (Listing 4). In this part, we can find two properties response time and server cost. The response time is represented by three states: low, medium, and high. In addition, the server cost is represented by two states: budget is under or over.

```
package bindingPart;
import znn.*;
import client.*;

abstract public cclass RTHigh extends
ZnnArch {
  public  cclass  RequestClient  extends
Client wraps InfoRequest {
  public float resptime () {
   return 5;}}}
```

Listing 4: Binding part of Znn.com.

The *Weavlets* are implemented through an empty CaesarJ class that extends the *Implementation* and the *Binding*. These *Weavlets* are deployed in the application by "*Weavlet Deployer*".

6.2 Evaluation

In this section, we evaluate the proposed approach using two criteria: reusability and energy consumption.

6.2.1 Reusability

Reusability refers to the degree to which existing applications can be reused in new applications. In this property, we compare a Java and CaesarJ implementation of Znn.com application with some criteria: separation of concerns (with two attributes: concern diffusion over components **CDC** and concern diffusion over operations **CDO**), coupling (with coupling between modules attribute **CBM**) and cohesion (with lack of cohesion attribute **LCO**) (Cacho et al., 2006).

CDC counts the number of modules whose main purpose is to contribute to the implementation of a concern and the number of other classes and collaboration interfaces that access them. **CDO** counts the number of operations whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them. **CBM** counts the number of modules to which a module is coupled. **LCO** counts the number of operations within a module that does not access the same instance variable (Cacho et al., 2006).

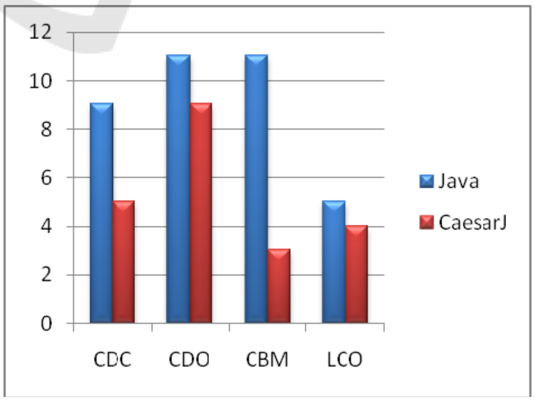

Figure 8: separation of concerns, coupling and cohesion in Znn.com application.

The implementation of Znn.com application with CaesarJ presents better results in terms of the number of diffusion over components (**CDC**), the

number of operations whose main purpose is to contribute to the implementation of a concern (**CDO**) and the number of modules to which a module is coupled (**CBM**).
In addition, the number of operations within a module that does not access the same instance variable (**LCO**) is not reduced (Figure 8).

6.2.2 Energy Consumption

This property represents the amount of energy consumed in a system. We evaluate the energy consumption in the client machine (with laptop) by two properties: screen (by brightness) and battery.

We consider the battery level before adaptation in 92% (175 minutes remaining). The level of the battery is decreasing to the level 7% after 124 minutes without adaptation (Figure 9).

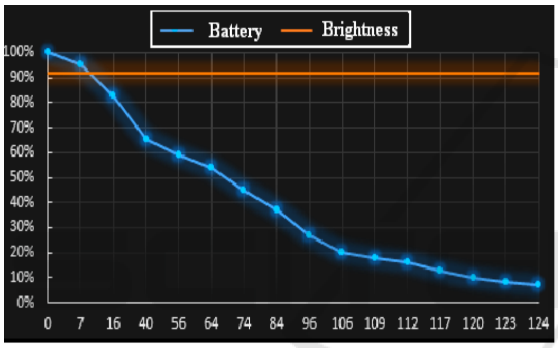

Figure 9: Level of battery without adaptation.

After the adaptation, the level of the battery is decreasing to the level 7% after 157 minutes. We have a gain of 33 minutes (Figure 10).

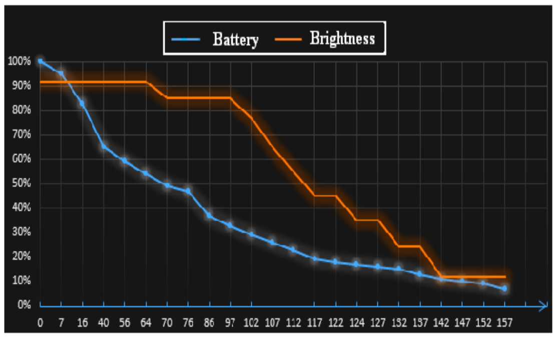

Figure 10: Level of battery with adaptation.

7 RELATED WORK

This section overviews selected efforts conducted by researchers to facilitate the modelling of elasticity.

(Buyya et al., 2010) propose an architectural for utility-oriented federation of Cloud computing environments. They provide a federated cloud infrastructure approach to represent elasticity for applications. However, their solution does not separate the components of architecture. In our case, we have used MAPE-K component to represent the components of system.

The authors in (Vaquero et al., 2011) present an approach that manages the elasticity with both a controller component and a load balancer one. However, the authors do not clearly present the behaviour of elasticity. In our solution, we have combined the feature model with state transition diagram into a single solution to show the behaviour of elasticity system.

(Marshall et al., 2010) develop a model that adapts services provided within a site, such as batch schedulers, storage archives, or Web services to take advantage of elastically provisioned resources. However, their approach does not support adding additional resources managers. With the extensibility of our approach, we can easily add new resource managers (functions or components in horizontal axis or configuration in vertical axis).

(Paraiso et al., 2014) present an approach that support portability, provisioning, elasticity, and high availability across multiple clouds. The authors use the annotations notion to express elasticity rules that ensure the appropriate decisions. In addition, (Berkane et al., 2015) propose an approach for developing self-adaptive systems at multiple levels of abstraction. This approach allows the combination of variability with feature model and reusability with design pattern into a single solution. In our solution, we have used the feature model and MAPE-K components to express the elasticity. The rules have modelled by using the feature model with state transition diagram.

8 CONCLUSION

In this paper, we have presented an approach for modelling elasticity system for cloud computing in a modular way. Based on self-adaptive systems and feature modelling, our approach supports the modularity of applications in structural and behavioural phases. The first phase consists to provide the skeleton for the overall structure of the application by separating it into distinct components based on functions of the application and MAPE-K architecture. In the second phase, we have combined the feature model with state transition diagram into a single solution to represent the different configurations of applications. As for future work, we are considering the dynamic variability of feature model in systems. This dynamic variability permits to represent the system at run-time (during execution). We will also try to specify the elasticity in a formal way by redefining the constraint of system. This formal specification will open the door to the reuse of features in the different system contexts.

REFERENCES

Bahga, A., Madisetti, V., 2013. *Cloud Computing: A Hands-On Approach*, CreateSpace Independent Publishing Platform.

Berkane, M.L., Seinturier, L., Boufaida, M., 2015. Using variability modelling and design patterns for self-adaptive system engineering: application to smart-home. *In International Journal of Web Engineering and Technology*, 2015 Vol.10, No.1, pp.65

Buyya, R., Ranjan, R., Calheiros, R., 2010. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 13–31.

Cacho, N., Sant Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C., 2006. Composing design patterns: a scalability study of aspect-oriented programming. In the *5th International Conference on Aspect-Oriented Software Development*, pp.109–121, ACM.

Cheng, S.-W., Garlan, D., Schmerl, B., 2009. Evaluating the effectiveness of the Rainbow self-adaptive system, *In 4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, IEEE.

Czarnecki, K., Eisenecker, U., 2000. *Generative Programming Methods, Tools, and Applications*, Addison-Wesley, Boston, MA, USA.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, Vol. 10, No. 1, pp.7–29.

Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J., 2004. *Software Factories: Assembling Applications with Patterns, models, Frameworks and Tools*, John Wiley & Sons, New York.

Herbst; N. R., Kounev, S., Reussner. R., 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. *In the 10th International Conference on Autonomic Computing* (ICAC 2013), San Jose, CA, June 24–28.

IBM, 2006. An Architectural Blueprint for Autonomic Computing, Technical Report.

Marshall, P., Keahey, K., Freeman, T., 2010. Elastic site: Using clouds to elastically extend site resources. *In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 43–52. IEEE Computer Society.

Mell, P., Grance, T., 2011. *The NIST Definition of Cloud Computing*, National Institute of Standards and Technology, U.S. Department of Commerce, NIST Special Publication 800-145.

Nunez, A., Gasiunas, V., 2015. ECaesarJ User's Guide http://ample.holos.pt/gest_cnt_upload/editor/File/public/ECaesarJ-manual.pdf [online], 2009, (accessed 2 March 2015).

Paraiso, F., Merle, P., Seinturier, L., 2014. soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds, *In Springer Computing, Springer*.

Vaquero, L., Rodero-Merino, L., Buyya, R., 2011. Dynamically scaling applications in the cloud. *In ACM SIGCOMM Computer Communication Review* 41(1), 45–52, 2011.