A Component-Based Adaptation Approach for Multi-Cloud Applications

André Almeida[1,2], Everton Cavalcante[2], Thais Batista[2], Nelio Cacho[2], Frederico Lopes[2]
[1]Federal Institute of Education, Science and Technology of Rio Grande do Norte, Parnamirim, Brazil
[2]Federal University of Rio Grande do Norte, Natal, Brazil
andre.almeida@ifrn.edu.br, evertonrsc@ppgsc.ufrn.br, thais@ufrnet.br, neliocacho@dimap.ufrn.br, fred@imd.ufrn.br

***Abstract*—This paper presents a dynamic adaptation process for multi-cloud applications that is composed of two phases. The first one is a modeling phase, which borrows the ability for representing commonalities and variabilities from the software product lines (SPL) paradigm. In this phase, a customized model feature specifies the configuration of cloud services to be used by an application (commonalities) and the different possible providers for each service (variabilities). The second phase is an implementation phase, which relies on the Mape-K autonomic loop to define the steps of an adaptation strategy and a centralized knowledge management provides the proper inputs for each step. Finally, in order to make easier the task of developing the adaptation process itself, the current implementation of the adaptation strategy relies on the FraSCAti component framework, which allows performing wiring and unwiring operations between components.**

***Index Terms*—Multi-Cloud Applications, Component-Based Dynamic Adaptation, Mape-K Autonomic Loop, FraSCAti.**

I. Introduction

With the advances in the Cloud Computing paradigm and the existence of several available offers of computing services in a pay-per-use model, some applications can be composed of multiple services provided by distinct, heterogeneous third-party cloud providers. Due to the facilities of acquisition and management of such services, developers can focus on application's logic rather than on processing power, data storage limitations or software infrastructure.

In this scenario, cloud services may undergo instability and Quality of Service (QoS) fluctuations during the execution of such multi-cloud applications. Therefore, it is required to adapt them in a dynamic and seamless way and with minimal user intervention in order to replace a given service by another one with better options for QoS, price or availability. When performing the adaptation process, it is necessary to select which cloud platforms/services must be used according to user-defined requirements and/or other constraints. However, the support for such an automatic adaptive behavior is not trivial as it involves the ability of changing cloud services that compose an application. Moreover, developers need to implement the adaptation code that deals with the replacement, which often is a time-consuming and error-prone task.

This paper presents a dynamic adaptation process for multi-cloud applications that is composed of two phases. The first one is a *modeling phase*, which borrows the ability for representing commonalities and variabilities from software product line (SPL) paradigm [1]. In this phase, a customized feature model [1] specifies the configuration of cloud services to be used by an application (*commonalities*) and the different possible providers for each service (*variabilities*). The second phase is an *implementation phase*, which relies on the Mape-K autonomic loop [3] to define the steps of an adaptation strategy and a centralized knowledge management provides the proper inputs for each step. Finally, in order to make easier the task of developing the adaptation process itself, the current implementation of the proposed adaptation strategy relies on the FraSCAti component framework [8], which allows performing wiring/unwiring operations between components.

Considering that each cloud platform/service has its own API, the use of a component-based approach allows the separation of each service/platform into a specific component that deals with the corresponding API. Each component is designed to be independent from the base application, so that any new platform/service added to the application does not compromise the original application. Since the Cloud Computing scenario is dynamic, it is imperative to rely on a solution that allows the dynamic reconfiguration suited to this scenario. As previously mentioned, FraSCAti has such dynamic reconfiguration capabilities and it was also evaluated in a cloud scenario [8].

This paper is structured as follows. Section II presents the background of this work. Section III introduces our approach for supporting the adaptation of multi-cloud applications based on the Mape-K loop and on the FraSCAti framework. Section IV analyzes related works. Section V contains final remarks.

II. Background

A. Software Product Lines

Software product lines (SPLs) [1] support the derivation of a wide range of applications by promoting the systematic and strategic reuse of components and other software artifacts. SPLs have become a mainstream technique to the development of software systems that share a common set of *commonalities* and contain *variabilities* that distinguish specific products, thus supporting the development of a *family* (or *product line*) of related products. Commonalities and variabilities between

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

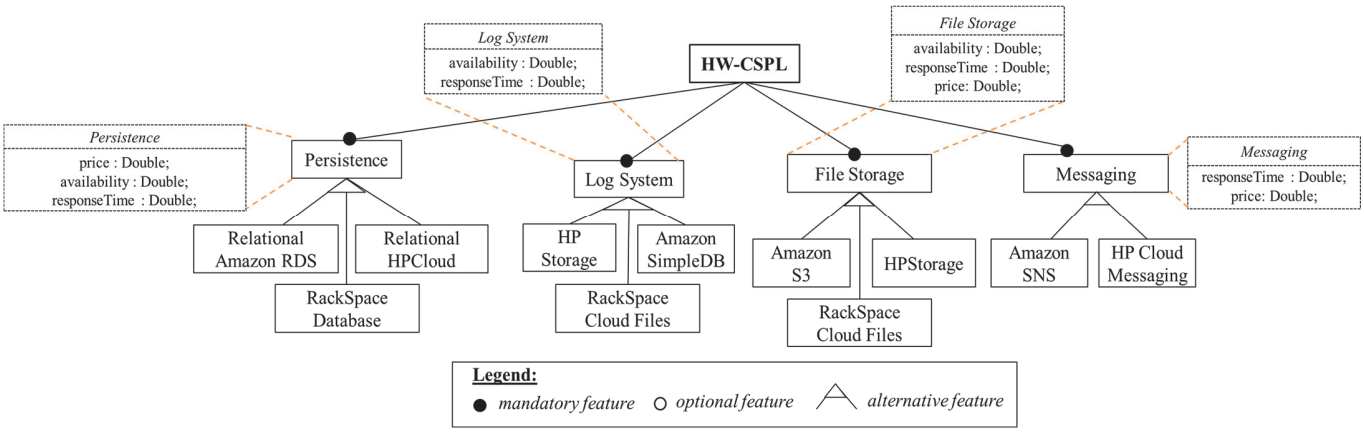

Fig. 1. HW-CSPL feature model.

products of a family are typically modeled in terms of *features*, which may be a requirement, a function, or a non-functional feature depending on the interest of such stakeholder involved in the application development. Features can be [6]: (i) *mandatory*, i.e., the feature must be included in a product; (ii) *optional*, i.e., the feature may or may not be included if the feature from which it derives is selected; (iii) *or-inclusive*, i.e., among the set of related features at least one of them must be selected, and; (iv) *alternative*, i.e., among the set of related features exactly one of them must be selected.

As described in our previous work [6], we have adopted an extended feature model in order to introduce *attributes* to features, in which an attribute is any characteristic of a feature that can be measured. These attributes are represented as <*name*, *type*, *value*> triples named *properties*. For instance, in a Cloud Computing environment, a property can represent any cloud-related information such as pricing, availability, response time and other QoS parameters of the provided service. These properties are also used for selecting the cloud services that fit the application needs. For example, consider that different cloud platforms offer a storage service and each one has different values for price and availability. Therefore, an application can specify its requirements in terms of price and availability of a storage service and our strategy is able to choose the specific cloud platform that offers a service that meets the established application requirements.

In order to illustrate such an approach, we have used HW-CSPL (*Health Watcher Cloud Software Product Line*) [6], an SPL developed from the *Health Watcher* (HW) [4] Web-based system. HW enables citizens to consult information about the public health system of a city and to register complaints in terms about contaminated food, diseases transmitted by animals, hygiene problems in restaurants, sewage leaks, etc. The commonalities were proposed from the requirements and features in the original HW system and the different service facilities provided by cloud platforms led to the features that represent the variabilities.

Fig. 1 illustrates the HW-CSPL extended feature model. It contains four mandatory features that represent commonalities: (i) *Persistence* is the persistence mechanism of the application; (ii) *Log System* is the infrastructure used for storing log information; (iii) *File Storage* defines how files (e.g., images related to the application data) are managed in the application, and; (iv) *Messaging* is a mechanism used to send messages via email, SMS and other notification services. Each one of these top-features has properties regarding the services represented by their alternative feature groups. For instance, the *Persistence* feature has three dynamic properties (*price*, *availability*, and *responseTime*) and offers three different options for persisting application data: (i) *Relational Amazon RDS*, which is related to the Amazon RDS database service provided by Amazon Web Services (AWS) platform; (ii) *Relational HP Cloud* feature, which is related to the relational database service provided by the HP Cloud platform , and; (iii) *RackSpace Database*, which is related to the database service provided by the RackSpace Cloud platform .

B. FraSCAti

FraSCAti [8] is an adaptive and reflective middleware platform for scalable heterogeneous multi-cloud systems that is based on Service Component Architecture (SCA) [5], thus allowing to describe components and connections between them and providing capabilities for dynamically wiring and/or unwiring components. FraSCAti also works as a standalone framework, so it is suitable to be deployed with a Web application, such as HW.

In FraSCAti, applications are built by composing distinct components (or services) and are independent from programming languages, interface definition languages (IDLs), communication protocols, and non-functional properties. FraSCAti also supports two types of adaptation: (i) *adaptation at design time*, in which developers can select the minimal set of features and then FraSCAti adapts the composition considering many different variants of execution environments, and; (ii) *adaptation at runtime*, in which FraSCAti is able to adapt the composition to new needs at runtime. The runtime adaptation can be applied for applications running on the platform, for non-functional services (e.g., transaction, security, etc.) used by some specific application, and for the container hosting application components.

The FraSCAti architecture has four main elements: (i) the *Component Factory* component is responsible for creating SCA components; (ii) the *Wiring & Binding Factory* component is responsible for dynamically creating wires between SCA components; (iii) the *Middleware Services* to be used by applications, and; (iv) the *Assembly Factory* component is responsible for parsing SCA assembly language descriptors, interpreting the XML tags and creating the corresponding SCA component assemblies. When it is necessary, the *Assembly Factory* component can: (i) call the *Component Factory* component for creating SCA components; (ii) call the *Wiring & Binding Factory* component for creating wires, exporting services and references, and; (iii) call the *Middleware Services* for integrating non-functional concerns into applications.

III. SPL-based Approach for Dynamic Adaptation of Multi-Cloud Applications

In this section, we present our SPL-based approach for supporting the deployment and dynamic adaptation of multi-cloud applications using SCA components provided by FraSCAti. Our solution enables developers to: (i) compose an application by using services provided by different cloud providers; (ii) change services at runtime as a reaction to environmental fluctuations without rewriting the application after its deployment, and; (iii) perform an adaptation process independently of programming technique and without user intervention.

As the main goal is to make an application adaptable to a multi-cloud scenario, it is necessary to consider how the already existing application code will be modified in order to support the adaptation process. In our previous work [6], we have used *dynamic aspect-oriented programming* (DAOP) for this purpose. However, such a programming technique may not be suitable for all types of applications and developers since they must have knowledge about the AOP paradigm, use third-party software to perform the dynamic weaving, and refactor the application to achieve the separation of concerns by using aspects. Based on such limitation, we have extended our previous work by adding the capability to support different programing techniques to enable dynamic adaptation without user intervention. In order to support this new feature, we have relied on the component-based technique supported by FraSCAti. Therefore, components that require services provided by other components (cloud platforms and services) can be wired/unwired at runtime.

Fig. 2 depicts our proposed approach for the dynamic adaptation of multi-cloud applications based on the Mape-K [3] autonomic control loop. Mape-K consists of four phases: (i) *Monitoring*, in which information is collected in order to be used for making the decision of adapting or not; (ii) *Analysis*, which defines if it is necessary to perform an adaptation; (iii) *Planning*, which establishes how the adaptation will take place and take any additional steps (e.g., migrating data), if required, and; (iv) *Execution*, in which the adaption process itself takes place. In order to support these phases, there is an element called *Knowledge*, which provides the required information. The following subsections describe how our approach implements the phases and elements of the Mape-K control loop.

A. Knowledge

This component is responsible for storing information used in the adaptation process encompassing the definition of which Cloud Computing platforms and their respective services can be used by the application. Such an information is represented by the *Variability Description*, which describes the services to be used by the application and their quality parameters and the possible options of cloud platforms (service providers) for each application service. The *Variability Description* is specified by a XML representation of the feature model and contains information about the relationship between commonalities and variabilities. Such a feature model description is defined without the need of specifying which programming technique is going to be used by the application in order to support the dynamic adaptation process. In turn, the *Implementation Configuration* is a XML file used for providing details about the implementation strategy to be used. In the case of a component-based implementation, such file contains the description of the components that require services (the common points) and the components that can provide these services (the variabilities). The variabilities deal with specific details of each cloud platforms/services in terms of APIs, programming models, etc.

B. Monitoring

For the *Monitoring* phase, there is an element called *Feature Monitoring Agent* that is responsible for monitoring the properties defined in the feature model regarding each Cloud Computing platform/service. For instance, as shown in Fig. 1, the *Log System* feature has two properties, namely *availability* and *responseTime*, and three associated variabilities, namely *HP Storage*, *RackSpace Cloud Files* and *Amazon S3*, which regards to the services provided by HP Cloud, RackSpace and AWS platforms, respectively. The values of these two properties are measured for the three platforms and then stored in the *Knowledge* component database in order to be used in the *Analysis* phase.

In order to monitor the defined properties, our monitor deploys dummy Web services, responsible for periodically invoke the services that might be used by the application. For instance, considering the *FileStorage* feature, three web services were created for each cloud platform and deployed on the same service. Such web services put a 20kbytes file into a storage unit in each cloud computing platform. During the periodically invocation of those web services, the properties are measured. Further details of the Monitor can be found at [11]

C. Analysis

In the *Analysis* phase, the *Decision Maker* component must select the configuration that better suits the user requirements in terms of the properties defined in the feature model. Such requirements are specified in terms of the *Decision Criteria*, which are composed of: (i) the *Selection Function*, composed

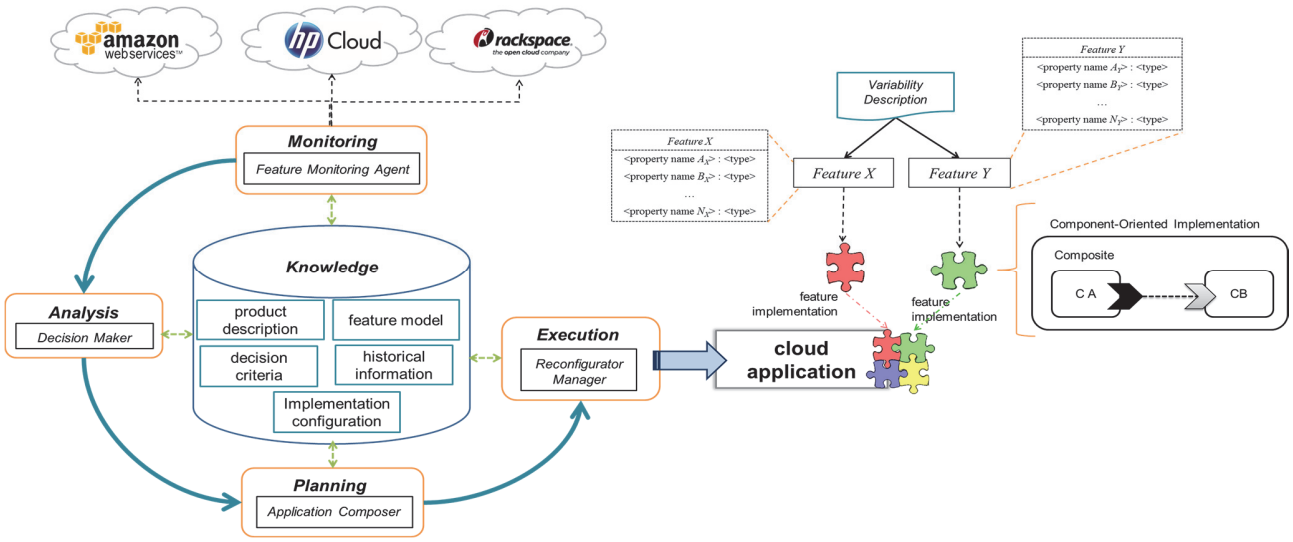

Fig. 2. Overview of our approach for the dynamic adaptation of multi-cloud applications based on the Mape-K autonomic control loop.

of functions defined by using the `max` and `min` functions and logical operator (`and`), and; (ii) the *Product Constraint Function*, which stands for a set of constraints specified by using comparison operators (`=`, `>`, `>=`, `<` and `<=`). For example, considering the feature model illustrated at Fig. 1, the *Decision Criteria* that select a configuration that maximizes the availability for the *File Storage* feature and minimizes the price for the *Persistence* feature, but also establishes that the response time for the *Log System* feature should not be greater than 10 ms, can be written as:

```
[max(FileStorage.availability) and
    min(Persistence.price)]
[LogSystem.responseTime < 10]
```

The selected configuration is described as a XML file called *Product Description*, which represents the product derived from the feature model and contains the name of the selected variabilities. This *Product Description* is used in *Planning* phase to generate the configuration required to make the adaptation process to work.

The *Selection Criteria* relies on user requirements in term of quality of service and other properties regardless of the cloud platform/service used. On the other hand, Service Level Agreements (SLAs) establish a contract between client and cloud provider, mainly in terms of quality of service that is going to be provided for the client. The later one is defined for each cloud platform/service and may vary in terms of what is actually delivered. This is different of the *Selection Criteria* of our work that specifies policies to select the best configuration, independent of the already agreed SLA between the client and the different cloud platforms/services.

D. Planning

The *Planning* phase is a preparation phase in which the *Product Description* is transformed in order to support the adaptation process. The artifact generated from the *Product Description* is called *Artifact for Dynamic Adaptation*, which can be a SCA composite description in case of using such component component-based technique. Depending on the chosen technique, the contents of this file are according to the tools used by the *Reconfiguration Manager* at the *Execution* phase (see Section III.E). In the case of the FraSCAti framework, it is generated a *composite*, which describes the components and the connection between them in order to compose the application and it is managed at runtime for a dynamic reconfiguration. Fig. 3 presents the steps of the *Planning* phase in order to generate the *Artifact for Dynamic Adaptation* required in the *Execution* phase to reconfigure the application. The *Application Composer* component is responsible for transforming the *Product Description* generated by the *Decision Maker* component into an *Artifact for Dynamic Adaptation*, which stands for a XML description managed by the *Reconfiguration Manager* for reconfiguring the application in the *Execution* phase.

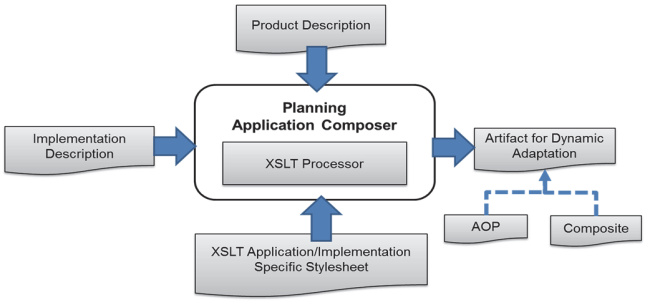

Fig. 3. *Product Description* transformation at the *Planning* phase.

The *Implementation Description* is an XML file containing all the definitions required by *Reconfiguration Manager* to deal with the process of the reconfiguration. In a component-based adaptation, it consists of the definition of components and the possible implementations related to the variabilities described in the feature model. Fig. 4 shows a fragment of the XML file

that defines the *PersistenceMechanism* component and the implementation associated with the possible variabilities. In line 3, the *basecomponent* tag defines a component called *PersistenceMechanism*, which has an interface named *IPersistenceMechanism* and it is exposed as the *Persistence* service. Lines 4 to 6 define a possible implementation called *RelationalAmazonRDS*, which has the name of the variability contained in the defined feature model and the class responsible for implementing it. Lines from 7 to 12 follow the same principle for the *RelationalHPCloud* and *RackSpaceDatabase* variabilities. Although the presented configuration uses simple principles from SCA, the developer is free to use full capabilities of SCA descriptions. For this, (s)he needs to update the *Implementation Description* and the *XSLT Application/Implementation Specific Stylesheet* file, which uses XSL Transformations (XSLT) in order to generate the composite file used by *Reconfiguration Manager*. By using XSLT, our solution remains implementation-agnostic, thus making possible to use any kind of implementation strategy until this phase of the control loop.

```
1. <configuration>
2.    <basecomponent name="PersistenceMechanism"
3.       interface="IPersistenceMechanism" service="Persistence">
4.       <implementation name="RelationalAmazonRDS">
5.          healthwatcher.frascati.persistence.PersistenceAWS
6.       </implementation>
7.       <implementation name="RelationalHPCloud">
8.          healthwatcher.frascati.persistence.PersistenceHP
9.       </implementation>
10.      <implementation name="RackSpaceDatabase">
11.         healthwatcher.frascati.persistence.PersistenceRackSpace
12.      </implementation>
13.   </basecomponent>
14.   ...
15. </configuration>
```

Fig. 4. XML fragment of the *Implementation Description*.

E. Execution

Finally, in the *Execution* phase of the autonomic loop, the *Reconfiguration Manager* is responsible for reconfiguring the application according to the respective programming technique that was chosen. This component is implementation-specific since the nature of the programming technique chosen changes the way in which the reconfiguration process will take place.

Fig. 5 shows a modified version of a SCA diagram illustrating the new components added to the HW application in order to support the dynamic adaptation process. The composite description, called *Reconfiguration,* exposes the methods provided by the components as services. The current version of our work relies on a *Reconfiguration Broker*, which is a Java class responsible for receiving the requests made by the already existing code of the application to the components described at the composite. The calls to the *Reconfiguration Broker* are inserted into the already existing code of the applications by analyzing it, rewriting the calls made to the already existing services (such as *Persistence* and *Logging*), and implementing the new functionalities provided by the *Messaging* and *FileStorage*. The implementations for the *PersistenceMechanism* component are described as *P1*, *P2* and *P3*, which can be Java class(es) that implement(s) persistence facilities by using Amazon, HP and RackSpace services.

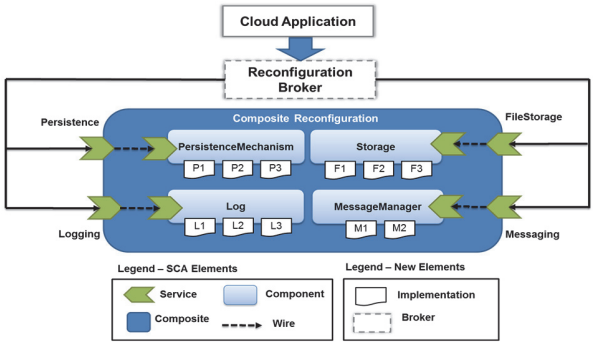

Fig. 5. Modified SCA diagram.

Fig. 6 shows a code fragment from the *healthwatcher.view.command.InsertSymptom* class, which is responsible for storing a disease symptom description with a photo to illustrate such symptoms. In line 1, a reference from the submitted file is created. In line 2, the *Reconfiguration Broker* is used to retrieve the current implementation of the *Storage* service, which is the way in which FraSCAti exposes the components located within a composite. The *IStorage* interface contains the *storePhoto* method, which receives as parameters the code of the photo and current file to be stored. This interface is recognized by the FraSCAti framework through Java annotations (used in our current implementation), which informs to the framework that such interface exposes a service provided by a component/composite. Finally, the *Storage* service is called to store the photo file into the current selected storage provider (line 3).

```
1. File photo = new File(request.getInput("photo"));
2. IStorage store =
        ReconfigurationBroker.getService("Storage", IStorage.class);
3. store.storePhoto(code, photo);
```

Fig. 6. Execution of the *Reconfiguration Broker*.

FraSCAti also allows the XML specification of which interfaces expose the services. If the developer wants to avoid changes in the original source code, (s)he needs to reflect these changes into the *Implementation Configuration* and the XSTL rules, as described in Section III.D.

The source code and samples of the XML files are available at http://www.dimap.ufrn.br/splmonitoring/adaptmcloud. In this URL, the reader can also find some evaluation experiments conducted in this work, mainly regarding monitored data and performance related to changing of configuration and the time required to reconfigure an application.

IV. Related Work

In our previous work [6], we have used DAOP as programming technique for writing the code to change the application base code. Such operations are based on the insertion/removal of aspects at runtime through the JBoss AOP framework [7], which supports the (re)definition of aspects by using XML descriptions and/or Java annotations. The dynamic weaving is achieved through bytecode instrumentation by relying on privileged accesses to the Java Virtual Machine (JVM), which may be not possible in constrained cloud

environments (for instance, the Google App Engine platform does not support bytecode instrumentation). In turn, as presented in this paper, the component-based approach developed on the top of the FraSCAti framework supports the redefinition of component implementations or even rewiring components without special JVM permissions.

Paraiso et al. [8] present a federated multi-cloud PaaS infrastructure that uses FraSCAti to deploy SCA applications on multiple heterogeneous PaaS and IaaS clouds. Similarly to our approach, the authors provide an infrastructure that allows the dynamic reconfiguration of component bindings and the addition of components and services at runtime. However, they have solely investigated the deployment of SaaS applications spanning different IaaS/PaaS and did not explore how an SPL could be used to support an automated selection of services to compose the application since such selection is manually performed in FraSCAti. It is also important to highlight that we use FraSCAti as a standalone library, thus enabling the dynamic reconfiguration using the composite description.

Ferry et al. [9] present the Cloud Modelling Framework (CLOUDMF) to build cloud-based dynamically adaptive systems (DAS) applications. The framework uses model-driven engineering (MDE) techniques in order to manage multi-cloud systems. Desair et al. [10] present a middleware platform for hybrid cloud applications that enables organizations to keep fine-grained control over the execution of their applications. The proposed middleware dynamically decides which requests and tasks are executed in a particular part of the hybrid cloud. Quinton et al. [2] present a model-driven approach based on feature models to handle cloud variability and then manage and create cloud configurations. They combine it with ontologies, which are used to model the various semantics of cloud systems. The approach considers application technical requirements as well as non-functional ones to provide a set of valid cloud or multi-cloud configurations and is implemented in a framework named SALOON. Our work is similar to these ones in the sense that we both investigate different approaches to model variability to support cloud configuration. However, our approach is more flexible as it supports an adaptation process in a seamless way and without user intervention. Furthermore, it is independent of programming techniques and provides advanced monitoring capabilities.

V. Final Remarks

This paper presented a dynamic adaptation process for multi-cloud applications managed by an autonomic control loop based on Mape-K [3]. The main phases of the control loop (*Monitoring*, *Analysis* and *Planning*) are implementation-agnostic, thus enabling developers to use any sort of programming technique and/or tools that support dynamic reconfiguration. One of these programming techniques is based on the definition of components that handle the specificities of the Cloud Computing platforms/services to be used by applications. The current implementation of such adaptation strategy was built on the top of the FraSCAti framework [8], whose dynamic nature and its known results have showed that it would be suitable to be used in our multi-cloud context. With FraSCAti, developers can focus on modularizing the already existing code into SCA components, which can (un)wired at runtime.

We also acknowledge the limitations and open issues associated with our work. One of the main problems is when dealing with services that are data-related. The cost of data migration must be taken into consideration when deciding to migrate from one cloud to another and also the need of maintaining the data integrity during the migration process. Another important issue is security. From authentication to security issues involved on migration of data, this is an important matter to be considered. However, these issues are out of the scope of this work since our main concern here is to present a solution that relies on a component-based framework to deal with multi-cloud platform and services. This solution provides an autonomous management cycle to deal with the reconfigurations and selection of cloud services.

In future works, we intend to extend our approach in order to consider information that can have influence on the dynamic adaptation process, such as the cost of a specific adaptation. In addition, it is important to provide support to high-level descriptions of the current *Decision Criteria*, thus enabling users to specify his/her policies in a simpler way. As the issues related with the migration of data/security are not tackled in this work, we intend to refine our solution to support third-party mechanisms that can provide solutions for these issues. Finally, we also intend to evaluate our solution by using not only different types of applications, but also a wider selection of Cloud Computing platforms.

References

[1] P. Clements and L. Northrop, Software Product Lines: Practices and patterns. USA: Addison-Wesley, 2001.

[2] C. Quinton et al., “Towards multi-cloud configurations using feature models and ontologies”, Proc. of the 2013 Int. Workshop on Multi-Cloud Applications and Federated Clouds. New York, USA: ACM, 2013, pp. 21-26.

[3] An architectural blueprint for autonomic computing, 4th ed. IBM, 2006 (Autonomic Computing White Paper)

[4] S. Soares, et al. “Distribution and persistence as aspects”, Software: Practice & Experience, 36 (7), Mar 2006, pp. 711-759.

[5] Beisiegel, M. et al., Service Component Architecture: Building systems using a Service-Oriented Architecture. IBM, 2005.

[6] A. Almeida et al., “Dynamic adaptation of Cloud Computing applications”, Proc. of the 25th Int. Conf. on Software Engineering and Knowledge Engineering.USA: 2013, pp. 67-72.

[7] JBoss AOP Framework: http://www.jboss.org/jbossaop

[8] F. Paraiso, et all., “A federated multi-cloud PaaS infrastructure”, Proc. of the 2012 IEEE 5th Int. Conf. on Cloud Computing. Washington, DC, USA, 2012, pp. 392-399.

[9] N. Ferry, et al. “Managing multi-cloud systems with CloudMF”, Proc. of the Second Nordic Symposium on Cloud Computing & Internet Technologies. New York, USA: ACM, 2013, pp. 38-45.

[10] T. Desair, et al., “Policy-driven middleware for heterogeneous, hybrid cloud systems”, Proc. of the 12th Int. Workshop on Adaptive and Reflective Middleware. New York, USA, 2013.

[11] A. Almeida et al., “Towards an SPL-based monitoring middleware strategy for Cloud Computing applications”, Proc. of the 10th Int. Workshop on Middleware for Grids, Clouds, and e-Science. USA: ACM, 2012.