Dynamic Adaptation of Cloud Computing Applications

André Almeida\textsuperscript{1,2}, Everton Cavalcante\textsuperscript{2}, Thais Batista\textsuperscript{2}, Nélio Cacho\textsuperscript{2}, Frederico Lopes\textsuperscript{2}, Flavia Delicato\textsuperscript{3}, Paulo Pires\textsuperscript{3}
\textsuperscript{1} Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN), Parnamirim, Brazil
\textsuperscript{2} Federal University of Rio Grande do Norte (UFRN), Natal, Brazil
\textsuperscript{3} Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil
andre.almeida@ifrn.edu.br, evertonrscc@ppgsc.ufrn.br, thais@ufrnet.br, neliocacho@dimap.ufrn.br, {fred.lopes, fdelicato, paulo.f.pires}@gmail.com

\textbf{Abstract}—Cloud-based applications are composed of services offered by distinct third-party cloud providers. As most cloud-related information (i.e., properties of the services such as price, availability, response time, etc.) of the services are dynamic and may change any time during the application execution, it is essential to adapt the application upon the detection of QoS violations that affect the application requirements. In this paper, we present a dynamic adaptation approach managed by an autonomic control loop that takes place when a service becomes unavailable or when QoS parameters are degraded. Our dynamic adaptation approach relies on dynamic aspect-oriented programming (DAOP) to: (i) encapsulate the dynamic adaptation (removal and/or insertion of services) as an aspect that contains join points that specify where each aspect must act, and; (ii) easily change the application by dynamically removing a service and inserting a new one.


I. INTRODUCTION

Cloud-based applications are inherently dynamic as they rely on a set of services provided by several underlying cloud computing platforms that can suffer from instability and Quality of Services (QoS) fluctuations. Moreover, the major difficulties in terms of developing such applications encompass issues such as the decision of which underlying cloud computing platforms to use, the need of continuously monitoring the dynamic cloud-related information of the very broad variety of services, and the need of adapting the application upon the detection of QoS violations that affect the application requirements. In this context, the software product lines (SPL)\textsuperscript{1} paradigm is useful for \textsuperscript{2}: (i) representing the alternative cloud services to be used by the applications as variabilities; (ii) configuring the application by choosing the proper cloud platform service that fits the application needs and; (iii) annotating the cloud-related QoS information as properties of each service, thus making it easier the identification of the dynamic information to be monitored.

In this paper, we present a dynamic adaptation approach for cloud-based applications that takes place when a service becomes unavailable or when an agreed QoS parameter is violated. Our dynamic adaptation approach relies on dynamic aspect-oriented programming (DAOP)\textsuperscript{4} to: (i) encapsulate the dynamic adaptation (removal and/or insertion of services) as an aspect that contains join points that specify where each aspect must act, and; (ii) easily change the application by dynamically removing a service and inserting a new one. In fact, the use of DAOP has been a natural choice for supporting adaptation of dynamic SPL\textsuperscript{5} as it supports late variability to deal with elements that can change at runtime, such as cloud services. In our scenario, the flexible variability mechanism of DAOP enables cloud services, represented as variabilities, to be woven and unwoven at the application at runtime. In addition, we use our existing SPL-based monitoring strategy\textsuperscript{3} to detect when an adaptation is required.

This paper is structured as follows. Section II briefly presents the background of this work. Section III contains the description of our adaptation approach for Cloud Computing applications, and the implementation details. Section IV presents an evaluation of our approach. Section V discusses related works. Finally, Section VI presents final remarks.

II. BACKGROUND

A. Cloud Computing

Cloud Computing is a paradigm that enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or interaction with the service provider and are provided in a pay-per-use way to the user. There are three fundamental models of cloud service providers: (i) IaaS (Infrastructure as a Service) platforms, which often provide physical resources such as virtual machines, servers, networks, etc.; (ii) PaaS (Platform as a Service) platforms, which typically provide an underlying infrastructure to develop, deploy, execute, and manage cloud-based applications, and; (iii) SaaS (Software as a Service) platforms, which provide application software on the cloud and users can access them by using a browser.

Building cloud-based applications is a challenging task as they are significantly more complex due to the intrinsic complexity of using third-party cloud providers. The major difficulties encompass issues such as the decision of which underlying Cloud Computing platforms to use, and the need of tracking pricing policies of services provided by different clouds platforms, thus hampering the development of applications using different cloud services\textsuperscript{2}. Furthermore, the particular nature of Cloud Computing applications creates specific requirements that also demand changes in terms of the development of such applications, encompassing methodologies and techniques for requirements elicitation, architecture, implementation, deployment, testing, and evolution of software. In such context, this
paper strives to make easier the development of applications using different cloud services, so that these applications can be monitored at runtime and adapted under dynamic conditions that may affect their requirements. To achieve these goals, our approach uses two paradigms, namely software product lines [1] and aspect-oriented programming [7], as briefly introduced in Section II.B.

B. Software product lines and Aspect-Oriented Programming

SPL [1] enable the creation of a family (or product line) of similar products by identifying commonalities (similarities) between all members of the family as well as characteristics 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.

AOSD [7] emerged as an approach to promote the modularization of crosscutting concerns, which are usually spread over several modules in a software system. Without proper means for separation and modularization, crosscutting concerns tend to be scattered over a number of modular units and tangled up with other concerns, thus resulting in lower cohesion and stronger coupling between modular units, and reduced degrees of understanding, evolvability, and reusability of software artifacts. In AOSD, crosscutting concerns are modularized as aspects, which are abstractions used to encapsulate crosscutting concerns that are associated with a set of classes and/or objects affected by such aspects. In turn, basic concerns of a software system are non-crosscutting concerns that can be modularized as conventional classes and objects, so that a mechanism called weaver is responsible for composing (weaving) the code regarding basic and crosscutting concerns. In aspect-oriented programming (AOP) [7] the major mechanisms to modularize crosscutting concerns in terms of aspects are join points, pointcuts, and advices. Join points are well-defined points in the application code (e.g. method calls) that specify how classes and aspects are related. Each aspect defines one or more expressions called pointcuts, which are used to select the join points that will be affected by the aspect’s crosscutting behavior. Finally, when the program execution reaches a join point selected by some pointcut expression, a piece of code called advice attached to a pointcut can be executed before, after or around it: (i) a before advice runs whenever a join point is reached and before the current computation proceeds; (ii) an after advice runs after the method body that has run and just before returning the execution control to the caller, and; (iii) an around advice runs whenever a join point is reached and has explicit control whether and when the computation under the join point is allowed to run.

In the synergetic relationship between AOP and SPL [8], recent research has pointed out that AOP promotes better modularity and changeability of SPL than conventional variability mechanisms [9]. Besides modularizing crosscutting concerns and managing variabilities in an SPL [10, 11], AOP is able to improve its evolvability and stability upon dynamic scenarios [11]. Aspects can contribute to modularize variabilities and facilitate their addition or removal according to the product (application) configuration, which is usually based on the selection of a set of features [8, 12]. For these reasons, we explore such relationship between AOP and SPL in our approach for supporting adaptation of an SPL as it supports late variability to deal with elements that may change at runtime [5], such as cloud services. As we present hereafter, the flexible variability mechanism of dynamic AOP enables cloud services, represented as variabilities, to be woven and unwoven at the runtime, thus supporting the adaptation of the application [4].

C. Running example: HW-CSPL

In our previous work [2], we have proposed a seamless adaptation of the SPL-based development to support specificities of cloud-based applications by adopting an extended feature model in order to introduce attributes to the features, in which an attribute is any characteristic of a feature that can be measured. Similarly, we also ground on this idea of introducing attributes to features in the feature model with the notion of properties, which have the form of <name, type, value> triples regarding a feature. Thus, a property can represent cloud-related information such as pricing, elasticity support and QoS parameters. In addition, the feature model becomes more expressive in order to represent important characteristics of the cloud services such as pricing model, availability, and response time. In order to illustrate such approach, we have developed HW-CSPL (Health Watcher Cloud Software Product Line), an SPL developed from the Health Watcher (HW) [13] real Web-based system. HW enables citizens to consult information about the public health system of a city and to register complaints in terms: (i) ingestion of contaminated food; (ii) mistreatment of animals or diseases transmitted by contaminated animals; and; (iii) other cases, e.g. 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 mandatory features representing commonalities: (i) Persistence, the persistence mechanism of the application, and; (ii) Log System, the infrastructure used for storing log information. Such model also contains one optional feature, File Storage, which defines how files (e.g. images related to the application data) are managed in the application. 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 two options for application’s data persistence, respectively represented by the Relational Amazon RDS feature, which is related to the Amazon RDS database service provided by Amazon Web Services (AWS) [18], and by the Relational HP Cloud, which is related to the relational database service provided by the HP Cloud platform [19].

III. Dynamic adaptation of cloud applications

QoS parameters and other dynamic-kind information regar-
Figure 1. HW-CSPL feature model.

Figure 2. Overview of the proposed dynamic approach.

 ding the used cloud services may change over time, thus affect-
ing the deployed applications that make use of such services. In
this perspective, our previous work [3] introduced a strategy
that enables to continuously monitor the dynamic properties of
the cloud services that are required/used by an application. In
this work we extend our previous approach by using the
MAPE-k loop [25], as illustrated in Fig. 2. In the Monitoring
phase the values gathered by the Feature Monitoring Agent are
stored in a database managed by the Knowledge component,
which is currently responsible for storing all information used
in our strategy to achieve the adaptation of cloud applications.
In the Analysis phase, the Product Generator and Evaluator
generates the product description, which stands for the config-
uration of the product to be deployed. This is achieved by pars-
ing the feature model and evaluating the product selection
criteria, described in more details in our previous work [3].
Afterwards, the generated product description is stored in the
Knowledge component and serves as input for the Planning
phase, in which the Aspect Composer component parses the
product description in order to generate a pointcut/advice mod-
el to be used by the Dynamic Weaver component in order to
reconfigure the application in the Execution phase. All of these
elements are conceptually described in the following subsections.
Although Cloud Computing offers several models, we
concentrate our approach on IaaS platforms, more specifically
the AWS and HP Cloud platforms, but our strategy is generic
and can be used with other platforms. As we are working with
dynamic adaptation using QoS information, it is fair to com-
pare services provided by platforms that follow the IaaS model.

A. Feature model

In our previous work [3], the feature model regarding the
SPL was extended in order to enable the user to annotate the
features with dynamic properties to be monitored. Now, such
feature model was extended again by adding two new ele-
ments: (i) the points of interest (represented as pointcuts) that
describe which parts of the application are susceptible to adap-
tation; (ii) the code responsible for implementing the variability
(represented as an aspect) and how they are bound to the
pointcuts.

Fig. 3 shows a fragment of the XML representation regard-
ing to the FileStorage feature in HW-CSPL. In lines 2 to 6 in
Fig. 3, the pointcuts tag contains the declaration of the pointcut
pc01 associated to the FileStorage feature (lines 3 to 5). In line
4, the pointcut expression execution(hw.Storage->store(*))
means that the interception must happen when the store meth-
od regarding the Storage class is executed even if it has multi-
ple signatures (as represented by the wildcard *). In addition,
it is possible to define how such pointcuts are associated to the
aspects/advises that implement the variabilities of the feature
model by using the bindings tag, as shown in lines 9 to 12 of
Fig. 3. Advice types are the same used on traditional AOP: in
a before advice a specific behavior must be executed before
reaching the pointcut, in an after advice, a specific behavior
must be executed after reaching the pointcut, and in an around
advice replaces the current execution of the pointcut. As
shown in lines 10 and 11, the pointcut pc01 is related to an
around advice named store and that is implemented by the
class hw.aspects.storage.HPStorage class.

B. Product Generator and Evaluator

The extended feature model and the product selection cri-
teria serve as inputs to the Product Generator and Evaluator
component, which was extended from our previous work [3].
This component evaluates the product selection criteria by
using the monitored values of the feature attributes and then
generates the product description, a XML description of the
selected product according to such criteria and that will be
deployed/adapted. Such product description consists in speci-
face to receive the
ed by the
model
the first deployment of the application, the
product description in order to generate a
pointcuts expressions and their
fying the features that compose the product and contains
information about the pointcuts and a reference to the implementa-
tion of the variabilities that compose the product, as well as
where aspects must be weaved into the application, thus en-
compassing the definition of pointcuts expressions and their
associated advices and aspects.

Fig. 4 shows a fragment of the XML representation of the
product description that uses the RelationAmazonRDS variability
associated to the Persistence feature. In lines 3 to 13, the
pointcut and aspect/advice are described for the
RelationalAmazonRDS variability. Due to space restrictions, this description
contains only the information needed for reconfiguring the application, so that the other details specified in the XML rep-esentation of the feature model (see Fig. 3) were removed.

For dynamic weaving capabilities, our Dynamic Weaver component must: (i) provide a grammar that enables to define
pointcut expressions or the capability for adding such expres-
sions; (ii) provide an API that supports the runtime definition of
aspects and how pointcuts are related to advices/aspects, and;
(iii) use a code instrumentation strategy to support dynamic weaving of aspects. The Dynamic Weaver implementation used in this work is the JBoss AOP framework [16], which supports static and runtime weaving. It enables to insert aspects by using:
(i) annotations or XML annotations for static changes, and;
(ii) the JBoss AOP API for runtime changes. Our pointcut
description uses the full grammar of JBoss AOP, thus enabling
the user to define a rich feature model since the feature imple-
mentation (variability) may be scattered in multiples point of
the code. To support the runtime binding/unbinding of aspects,
JBoss AOP relies on code instrumentation, thus incurring in the
manipulation of the generated Java bytecode. Many other solu-
tions for dynamic weaving, such as PROSE [22] and As-
pctWerkz, rely on the modification of the Java Virtual Ma-
chine (JVM), which may be a prohibitive issue in the context of
Cloud Computing applications. However, a drawback of JBoss
AOP is that it must be loaded with the JVM used by the con-
tainer/application server, which is required by the Java pro-
gramming language to instrument the Java code. Since it re-
quires more access to the JVM, using such solution for a PaaS

```xml
1. <product>
2.  <feature name="RelationalAmazonRDS">
3.    <pointcut>
4.      <pointcut name="ppeer">
5.        execution(lib.persistence, PersistenceMechanism.class, getcommunicationChannel())
6.      </pointcut>
7.    </pointcuts>
8.  </feature>
9. </product>
```

Figure 4. XML representation of the product description.

C. Aspect Composer

The Aspect Composer component is responsible for parsing the product description in order to generate a pointcut/advice model, which is a representation of the product to be adapted with pointcuts, aspects, and advices and described how the application must be reconfigured by using the interface provided by the AOP Handler component, which provides an inter-
face to receive the pointcut and aspect/advice models. If it is
the first deployment of the application, the AOP Handler is
accessed to define the pointcuts and bind the aspects/advices to
such pointcuts. If the application has already been deployed, it is
necessary to first unbind the current aspects associated to the
application and then bind the new aspects that are described in the
product description.

The AOPHandler component is responsible for interacting with the Dynamic Weaver implementation, so that it is neces-
sary to know how it works. Our proposal is to provide a flexi-
ble solution in order to enable the developer/user to change the
Dynamic Weaver component. For this purpose, we defined an
AOPHandler interface to be used by the Aspect Composer to
register/unregister the pointcuts, aspects, and advices. Our
current implementation uses the JBoss AOP framework [16]
for implementing the Dynamic Weaver component, so that the
AOPHandler component knows how to deal with the provided
API. For example, if the developer/user wants to change from
JBoss AOP to the Guice injection framework [17], the provided
implementation must know how to: (i) register the defined
pointcuts by interpreting the pointcut grammar, and; (ii)
bind/unbind pointcuts with the defined aspects/advices.

D. Dynamic Weaver

The Dynamic Weaver component supports not only the
AOP programming model, but it also has the capability of
dynamically weaving code into the application. In traditional
AOP, the developer must specify, at design time, the
pointcuts and aspects (crosscutting concerns) that must be weaved into
the application. In traditional AOP, the weaving of aspects is
typically done only at compilation time. On the other hand,
dynamic weaving of applications stands for the capability of
dynamically introducing aspects within the application at runtime. To add new functionalities to components only avail-
able at the binary format, it is necessary to rely on code instru-
mentation, which means changing the bytecodes of compiled
classes of Java applications, for example.

For dynamic weaving capabilities, our Dynamic Weaver component must:
(i) provide a grammar that enables to define
pointcut expressions or the capability for adding such expres-
sions; (ii) provide an API that supports the runtime definition of
aspects and how pointcuts are related to advices/aspects, and;
(iii) use a code instrumentation strategy to support dynamic weaving of aspects. The Dynamic Weaver implementation
used in this work is the JBoss AOP framework [16], which supports
static and runtime weaving. It enables to insert aspects by us-
```xml
1. <alt abstract="true" name="FileStorage">
2.  <pointcuts>
3.    <pointcut name="pcpl">
4.      execution(Hw.Storage->store(*))
5.    </pointcut>
6.  </pointcuts>
7.  <feature mandatory="true" name="HFileStorage">
8.    <featuremonitoringbean="Monitoring.MonitoringHFileStorage">
9.      <bindings>
10.     <around pointcut="pcpl" aspect="Hw.aspects.storage.HFileStorage">
11.       name="store" />
12.     </bindings>
13.     </featuremonitoringbean>
14.   </feature>
15. </feature>
16. <feature mandatory="true" name="HFileStorage">
17.   <featuremonitoringbean="Monitoring.MonitoringHFileStorage">
18.     <bindings>
19.      <around pointcut="pcpl" aspect="Hw.aspects.storage.HFileStorage">
20.        name="store" />
21.     </bindings>
22.     </featuremonitoringbean>
23.   </feature>
24. </feature>
25. </feature>
26. </product>
```

Figure 3. XML representation regarding the FileStorage feature in HW-CSPL.
environment, which may have restrictions in terms of JVM manipulation, is not feasible. The Guice solution [17] is a dependency injector that does not require such access, but it lacks of support of full features for AOSD because it is necessary to implement all pointcut expressions and it only supports around advices. In future works, we intend to implement a solution that makes use of the Guice framework and evaluate our proposal with PaaS platforms.

IV. EVALUATION

We have evaluated the proposed approach by using the case study described in Section II.C. It was deployed in an OpenStack private cloud environment running a single cloud server instance (Tiny - 512 RAM and 1 VCPU) and using JBoss as its application server and Linux as base operating system. We have purchased an Amazon RDS instance and HP Compute instance with a MySQL database to evaluate the Persistence feature and loaded them with a set of random data. In order to evaluate the FileStorage feature, we have used the Amazon S3 and HP Cloud Storage services, and for the Logging System feature we have used the Amazon SimpleDB and HP Cloud Storage since the HP Cloud platform does not offer a non-relational database service like Amazon SimpleDB. In order to measure the performance of the whole adaptation process for adapting from a product to another, we have measured the time spent by the adaptation of each feature of the case study. As the product is selected based on the monitored values gathered from the purchased services, we have decided to provide a given set of feature monitoring values in order to trigger the adaption and to evaluate all possible scenarios (i.e. all the variabilities presented in the feature model). We have measured 1000 product changes by measuring the time for each feature takes to be adapted. In Table 1 we present for each feature of the SPL the maximum, minimum, and average times (in milliseconds) spent by the adaptation process.

### Table 1. Execution Time for the Dynamic Adaptation Process.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Minimum (ms)</th>
<th>Maximum (ms)</th>
<th>Average (ms)</th>
<th>Standard Deviation (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence</td>
<td>2988.36</td>
<td>3631.92</td>
<td>3240.19</td>
<td>194.67</td>
</tr>
<tr>
<td>File Storage</td>
<td>14.05</td>
<td>19.88</td>
<td>16.98</td>
<td>1.96</td>
</tr>
<tr>
<td>Log System</td>
<td>15.73</td>
<td>22.86</td>
<td>20.20</td>
<td>2.20</td>
</tr>
</tbody>
</table>

As shown in Table 1, the Persistence feature stands out when comparing with the other features due to the implementation of such feature. Since the HW system uses a pooling control for database connections, the adaptation process for this feature consists in releasing the connection pool and creating a new one according to the changed configuration (i.e. new URL, user and password for accessing the new database). The other two features are directly implemented to use the services (e.g. store an image by using the FileStorage feature or register a log entry by using the LogSystem feature), and since we are interested in evaluating the time for reconfiguring the application instead of the use of such services themselves, we consider such difference as a normal issue. Considering the set of changes and the observed standard deviations, for the Persistence feature we have only 27% of the 1000 test cases outside the standard deviation, and considering that lower times presents a better response, only 1/3 of the 27% (90 test cases) are above the average execution. For the FileStorage feature, the mentioned percentage is around 21% of the 1000 test cases. In this perspective, we consider that our solution presents a stable performance, considering the proposed scenario and feature model. It is important to notice that the measured time is added to the application execution time, thus meaning that the overhead due to the use of our strategy is represented by the times presented in Table 1. However, the process of adaptation is only triggered when changes in the deployed product occur, thus meaning that once a product is deployed and reconfigured, the application execution is not affected by our solution. It is also important to highlight that depending on the complexity of the features/variabilities, the overhead will be increased, as we mentioned before, but this time is counted for the feature implementation and not by our solution itself.

V. RELATED WORK

The idea of using SPL for supporting adaptation has been used by various works. Gomaa and Hashimoto [20] describe a dynamic software adaptation approach and environment for service-oriented product lines. This approach uses a dynamic feature model for a family of service-oriented architectures (SOA), in which a member of such family can be dynamically adapted to a different member of the family at runtime. The decision to adapt is based only on a set of software adaptation patterns, while our approach considers the Product Selection Criteria and the monitored values of the feature attributes. More similar to our approach, Baresi et al. [21] use the Common Variability Language (CVL) to augment BPEL (Business Process Execution Language) processes with variability. This makes it possible to generate a dynamic SPL and use an aspect-oriented based version of BPEL to manage and run the SPL. This approach is more suitable for self-adaptive SOA systems which are usually self-contained and loosely coupled. In contrast, our approach combines an expressive Product Selection Criteria with an extended feature model, which allows our approach to support a more fine-grained adaptation than the previous approach.

More recently, some approaches have been using SPL to manage software variability of cloud applications. In the Mietzner et al.’s work [14], variability techniques are used to support the management of variabilities in SaaS applications. Application templates describe the variability through variability descriptors. Likewise, FraSCAti [15], an adaptive and reflective middleware for multi-cloud systems, uses SPL to enable developers to select the configuration of the SaaS platform that matches the application needs. SPL is used only to represent the features and their constraints, which captures all possible configurations. In our work we go a step further by using SPL and aspects to implement an adaptation mechanism of cloud applications. This is an important difference of our work since our solution improves the adaptation and modularization of the application.

In addition, there has been much previous research using monitoring of QoS attributes to support dynamic adaptation. The Dai et al.’s work [23] is based on prediction of performance failures to support adaptation. The decision to adapt is based on the performance of a single service, while our approach considers the Product Selection Criteria and the moni-
tored values of each feature. In the Leitner et al.’s work [24], the PREvent approach is described to support prediction and prevention of SLA violations in service compositions based on event monitoring and machine learning techniques. The prediction of violations is calculated only at defined checkpoints in a composition based on regression classifiers prediction models. In contrast, our approach supports adaptations every time a product is chosen according to its Product Selection Criteria. Despite the similarities, none of these AOP approaches take the feature model and the Product Selection Criteria into account to support software adaptation.

VI. CONCLUSION AND FUTURE WORK

The development of cloud-based applications that are composed of services offered by distinct cloud providers is a hard task due to the inherent heterogeneity of cloud environments. The selection of the proper cloud services that fit the application needs is based on cloud-related information, which is used to triggering an adaptation process. In our previous work [3], we introduced a strategy that enables to continuously monitor the dynamic properties of the cloud services that are required/used by an application. If there is any change on the values of such properties that affects the requirements of the product already deployed, a dynamic adaptation process is triggered in order to make the application redeployment. In this paper we presented an adaptation strategy that relies on DAOPT to encapsulate the dynamic adaptation (removal and/or insertion of services) and easily change the application by dynamically removing a service and inserting a new one. The MAPE-K control loop enables to better define the phases of our strategy and a centralized knowledge management provides proper inputs to such phases. In addition, we are able to describe, in the feature model, the points in the application that must be intercepted to support the adaptation process. Moreover, it is necessary to provide a way for dynamically weaving aspects into the application and then changing it, based on the monitored values. For this purpose, we used the JBoss AOP framework, which has a rich pointcut expression grammar.

In future works, we intend to improve the algorithm that selects the best product to be deployed since it still generates and evaluates all possible products, which can be prohibitive in case of a larger SPL. Another important issue is to consider historical information in our strategy, for instance, how the cost of a specific adaptation affects a future adaptation. Finally, we also intend to evaluate other dynamic strategies and frameworks for handling dynamic weaving and dependency injection since the JBoss AOP project is not suitable for environments with runtime restrictions.

ACKNOWLEDGMENTS

This work was partially supported by Brazilian Academic and Research Network (RNP) through the AltoStratus Project, and by the National Institute of Science and Technology for Software Engineering (INES)\(^1\) funded by CNPq under grant 573964/2008-4. Thais Batista is partially supported by CNPq under grant 485935/2011-2, and Flavia Delicato and Paulo Pires are also partially supported by FAPERJ and CNPq (grants 311363/2011-3, 470586/2011-7, 310661/2012-9).

\(^1\) http://www.ines.org.br

REFERENCES