Leveraging Platform Basic Services in Cloud Application Platforms for the Development of Cloud Applications

Fotis Gonidis, Iraklis Paraskakis
South East European Research Centre
International Faculty of the University of Sheffield,
City College
Thessaloniki, Greece
{fgonidis,iparaskakis}@seerc.org

Anthony J. H. Simons
Department of Computer Science
The University of Sheffield
Sheffield, UK
A.Simons@dcs.shef.ac.uk

***Abstract*—** Cloud application platforms gain popularity and have the potential to alter the way service based cloud applications are developed involving utilisation of platform basic services. A platform basic service is considered as a piece of software, which provides certain functionality and is usually offered via a web API. However, the proliferation and diversification of platform basic services and the available providers increase the challenge for the application developers to integrate them and deal with the heterogeneous providers' web APIs. Therefore, a new approach of developing applications should be adopted in which developers leverage multiple platform basic services independently from the target application platforms. To this end, this paper presents a development framework assisting the design of service based cloud applications. The objective of the framework is to enable the consistent integration of the platform services, and to allow the seamless use of the concrete providers by alleviating the heterogeneities among them. The core components of the framework are the reference meta-model, which facilitates the modelling of the platform services and an ontology-driven architecture enabling the description and the abstraction of the providers' specific web APIs.

***Index Terms*—Platform Basic Services, Cloud Application Platform, Service-based Cloud Applications, PaaS, Cloud Computing**

I. INTRODUCTION

The rise and proliferation of cloud computing and cloud platforms in specific, has the potential to change the way we develop, distribute and consume cloud based service applications. Cloud platforms popularity stems from their potential to speed and simplify the development, deployment and maintenance of cloud based software applications. Nevertheless, there is a large heterogeneity in the platforms offerings [1] which can be classified into three clusters. On one cluster application development time is drastically decreased with the use of bespoke visual tools and graphical environments at the expense of a restricted application scope which is usually limited to customer relationship management (CRM) and office solutions. At the other end of the spectrum platforms offer basic development and deployment capabilities such as application servers and databases. The intermediate cluster consists of cloud platforms, which offer additional functionality via the offering of, what we call, *platform basic services* (eg mail service, billing service, messaging service etc). A platform basic service can be considered as piece of software which provides certain functionality and can be reused by multiple users. It is typically provisioned via a web API. The platforms offering such services are also referred to as *cloud application platforms* [2].

The rise of the *cloud application platforms* has the potential to lead to a paradigm shift of software development where the platform basic services act as the building blocks for the creation of service-based cloud applications. Applications do not need to be developed from scratch but can rather be constructed using, where appropriate, various platform services, thus increasing rapidly the productivity. Consequently, the barrier of studying the various platform basic services and selecting the one(s) best offered for the task at hand, is now removed. The software engineer has access in a transparent manner to all platform basic services and the selected platform basic services are seamlessly incorporated in the service based cloud application.

However, these opportunities are accompanied by a number of challenges. The first challenge arises from the fact that there exist multitudes of a particular service, e.g., mail service, since the services are offered by many different providers. The second challenge arises from the need to provide a framework that spans across a number of different kind of services, i.e., mail services, billing services, message queue services and so on.

The result of these two challenges implies that there exists a large heterogeneity among the offered services. The heterogeneity mainly arises due to (i) the differences in the workflow for the execution of the operations of the services, (ii) the differences in the exposed web APIs and (iii) the various required configuration settings and authentication tokens. The significant number of services that an application may consist of, makes the integration and management of the services a strenuous process. At the same time there is a lack of tools and Integrated Development Environments addressing the issue of proprietary technologies and APIs [3].

In order for the developers to be able to leverage *platform basic services* from various environments a new approach of application development should be adopted, where the latter are decoupled from specific platform technologies.

Towards this direction, the paper proposes a framework , that tackles the two aforementioned challenges, thus assisting the process of developing service-based cloud applications. The objective of the framework is two-fold: (i) First to enable the integration of platform basic services in a consistent way and (ii) second to facilitate the seamless use of the *platform basic service* providers by alleviating the heterogeneities among them. Thus application developers can focus on the design of the application without dealing with the peculiarities of each provider.

The framework adopts a three phase process in order to enable the abstraction of the platform service providers. First the abstract functionality of the platform basic service is described. During this phase the workflow of the platform service is modelled and the reference API is defined. In the next phase, the concrete vendor implementation is infused. The specific workflow and web API is mapped on the reference one defined in the first phase. During the third phase, the framework handles the execution of the workflow and automatically generates the client adapters to invoke the providers' web API.

The rest of the paper is structured as follows: Section II states the variability points that may arise among the platform basic services and which are addressed by the proposed framework. Next Section reviews the related work while Section IV lists the requirement of the framework. Section V presents the high-level design of the framework and the process of supporting additional platform services and providers. Section VI and VII describe in details the process of modelling the workflow and describing the API of the platform service respectively. Finally in Section VIII we conclude the paper and discuss future work.

II. Variability Points of platform basic services

Preliminary work of the authors on several platform service providers [4] offered by Heroku [5], Google App engine [6], AWS marketplace [7] have shown that the following three variability points needs to be addressed in order to decouple application development from vendor specific implementations: (i) Differences in the workflow for the execution of the operations offered by the platform basic services, (ii) variability in the web API exposed by the various platform service providers, (iii) management of the configuration variables and authentication tokens required during the interaction with the services.

1) Differences in the workflow: Stateful services require more than one state in order to complete an operation [8]. Such an example is the payment service that enables developers to accept payments through their applications. The process involves two states: (i) waiting for client's purchase request and (ii) submitting the request to the payment provider. However, depending on the concrete payment provider there may be variations in the states involved. Therefore, a coordination mechanism is required to handle the operation flow and additionally to alleviate the differences among the various concrete implementations.

2) Differences in the web API: There are several platform providers implementing a given platform service and specific operations. However, they expose a diverse API resulting in conflicts when an application developer attempts to integrate with one or another. As an example we consider the e-mail service and two service providers, the Amazon Simple E-mail Service (SES) [1] and the SendGrid[2], an add-on mail service offered via Heroku application platform. Upon the request for sending an e-mail the minimum set of the four following parameters are required: (i) the recipient, (ii) the sender, (iii) the content and (iv) the title of the e-mail. The concrete naming of the parameters as required by Amazon SES is respectively: (i) Source, (ii) Destination.ToAddresses, (iii) Message.Subject and (iv) Message.Body.Text whereas regarding the SendGrid the anticipated parameters are: (i) from, (ii) to, (iii) subject and (iv) text.

3) Management of the configuration and authentication variables: In addition to the construction of the web calls and the operation workflow handling, platform services require certain configuration settings and authentication tokens to be present during the interaction with the cloud application. Indicatively, we refer to the Google Authentication service (footprint) and the following set of required variables: a) the redirect URL, b) the client_ID, c) the scope and d) the state. The number and the type of the settings vary according to the provider. Considering the large number of services that an application may be composed of, the management of the settings may become a time consuming and strenuous process.

III. Related Work

The constant increase in the offering of platform services has resulted in a growing interest in leveraging services from multiple clouds. Significant work has been carried out on the field, which can be grouped into three high-level categories: middleware platforms, Model-driven Engineering techniques and library based solutions. Representative work on each of the three categories is listed.

Library-based solutions such as jclouds [9] and LibCloud [10] provide an abstraction layer for accessing specific cloud resources such as compute, storage and message queue. While, library-based approaches efficiently abstract those resources, they have a limited application scope which makes it difficult to reuse them for accommodating additional services.

Middleware platforms constitute middle layers, which decouple applications from directly being exposed to proprietary technologies and deployed on specific platforms. Rather, cloud applications are deployed and managed by the middleware platform, which has the capacity to exploit multiple cloud platform environments. mOSAIC [11] is such a PaaS solution which facilitates the design and execution of

[1] http://aws.amazon.com/ses/
[2] http://sendgrid.com

scalable component-based applications in a multi-cloud environment. mOSAIC offers an open source API in order to enable the applications to use common cloud resources offered by the target environment such as virtual machines, key value stores and message queues. OpenTOSCA [12], is a runtime environment enabling the execution of TOSCA-based cloud applications. TOSCA [13] is a specification which enables the description of the deployment topology of a cloud application in a platform independent way. Thus, applications are agnostic with regard to the concrete platform provider resources they use. Both mOSAIC and OpenTOSCA require that applications are tightly connected with the specific technologies and thus impose a restriction in case applications need to leverage platform providers, which are not supported by those environments.

Initiatives that leverage MDE techniques present meta-models, which can be used for the creation of cloud platform independent applications. The notion in this case is that cloud applications are designed in a platform independent manner and specific technologies are only infused in the models at the last stage of the development. MODAClouds [14] and PaaSage [15] are both FP7 initiatives aiming at cross-deployment of cloud applications. Additionally, they offer monitoring and quality assurance capabilities. They are based on CloudML [16], a modelling language which provides the building blocks for creating applications deployable in multiple IaaS and PaaS environments. Hamdaqa et al. [17] have proposed a reference model for developing applications which leverage the elasticity capability of the cloud infrastructure. Cloud applications are composed of CloudTasks which provide compute, storage, communication and management capabilities. MULTICLAPP [3] is a framework leveraging MDE techniques during the software development process. Cloud artefacts are the main components that the application consists of. A transformation mechanism is used to generate the platform specific project structure and map the cloud artefacts onto the target platform. Additional adapters are generated each time to map the application`s API to the respective platform`s resources.

The solutions listed in this Section focus mainly on the cross-deployment of application by eliminating the technical restrictions that each platform imposes. However, they do not support the use of additional platform services offered via web API such as payment, authentication and message queue service. In addition, the client adapters used to address the variability in the providers' APIs are hardcoded and thus not directly reconfigurable in case they are required to be updated. On the contrary, the vision of the authors is to facilitate the use of platform services from heterogeneous clouds in a seamless manner. To this end, the proposed solution attempts to alleviate the three variability points described in Section II, namely: the differences in the workflow modelling, in the providers' web APIs and in the configuration settings. In turn, this will promote the design of applications, which leverage services from multiple cloud application platforms without being bound to the specific proprietary implementations and APIs.

V. REQUIREMENTS OF THE FRAMEWORK

There are certain requirements identified for the development framework as listed below. They have primarily been defined upon the objective of addressing the variability points, which were listed in Section II, namely, the differences in the workflow, in the web API and the settings and tokens required by each concrete platform service provider.

- **Provide workflow modelling capabilities.** The development framework should provide application developers with the necessary building blocks to enable the workflow modelling in a consistent way. Independent of the type of the platform service or the concrete provider, two basic request types are present: (i) The outgoing request from the application to the platform service using the web API of the latter and (ii) the incoming requests usually by the platform service to the application which needs to be received and handled by the latter. The framework should enable the modelling of these request types.
- **Automating the execution of the workflow.** In addition to the modelling of the states of the platform service, an execution engine should be able to handle the operation workflow and thus decoupling the application developer from directly accessing the provider specific implementation.
- **Addressing the API variability.** In order to effectively abstract the vendor specific implementation, the framework should address the peculiarities in the various web APIs exposed by the concrete providers. Two further dimensions are implied: (i) The capability of defining a reference API for each given platform service and which is exposed to application developers. (ii) The mapping of the vendor specific API to the equivalent reference one.
- **Automatic generation of the client adapters.** The framework should be able to generate the code required to perform the invocation requests to the web API which is exposed by the platform service providers. The majority of the providers expose a web API, based on HTTP requests [18], and often adhere to the REST principles [19]. By offering code generation capabilities, the application developers are alleviated from the task to manually code the invocation request each time integration with a new service provider is required.
- **Generic nature of the framework.** One of the main requirements of the framework is its capability to support new platform services and providers. Rather than being static an rigid our objective is to ensure its flexibility so that it is continuously expanded and updated with new types of platform services and providers. This is partially achieved by the first and third requirement, namely by providing the generic building blocks to model the workflow of the platform service and also the capability of defining the reference API for the service which is supported by the framework.
- **Distinct user roles:** Two user roles should be supported by the framework: (i) the administrator and (ii) the

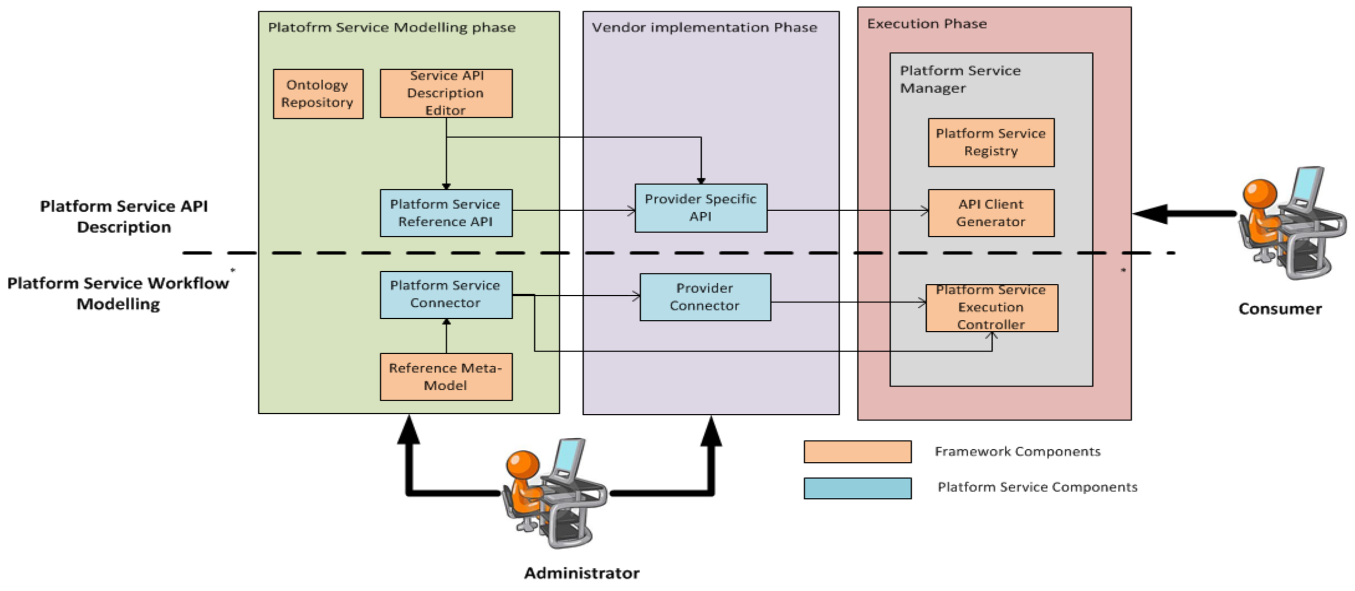

Platofrm Service Modelling phase
Ontology Repository
Service API Description Editor
Platform Service Reference API
Platform Service Connector
Reference Meta-Model
Vendor implementation Phase
Provider Specific API
Provider Connector
Execution Phase
Platform Service Manager
Platform Service Registry
API Client Generator
Platform Service Execution Controller
Platform Service API Description
Platform Service Workflow Modelling
Consumer
Administrator
Framework Components
Platform Service Components

Figure 1. High-level Overview of the Development Framework

consumer. The administrator should be capable of enhancing the framework with new platform services and providers. On the other hand, the consumer is the application developer who uses the services supported by the framework.

- **Management of the platform services and the configuration variables.** The framework should enable the application developers to add or remove services seamlessly from the application and also manage the configuration settings and the authentication tokens required by each of the concrete providers.
- **User Friendliness.** It is essential to ensure the ease of use of the framework. Therefore, an intuitive graphical environment should be designed and offered to the users so that the administrators can add new services and providers to the framework and the consumers can easily integrate or release services from the application.

In the next Sections we describe how the framework can be used to enable the integration of platform services and providers.

VI. High-Level Overview of the Framework

As it can be observed in Fig. 1, the process of adding a new platform service and provider to the framework can be divided into the following two parts:

i) Platform Service Workflow Modelling. Certain platform services require more than one step to complete an operation, such as the authentication and the payment service. Thus, the states that are involved in the execution of an operation shall be defined and modelled in a way that is capable for the framework to automatically handle the workflow.

ii) Platform Service API Description. One of the main objectives of the framework is to provide the developers with a single API for each platform service independent of the concrete provider. Therefore, this part involves the definition of the reference API, the description of the web API of each concrete provider supported by the framework and the subsequent mapping of the provider specific web API to the reference one.

For each of the two parts the development process involves the three following phases:

i) Platform service modelling phase. During this phase, the abstract functionality of the platform service is defined. Particularly, it requires the modelling of the states involved in each operation and the definition of the reference API that is exposed to the developers.

ii) Vendor implementation phase. Based on the abstract model defined in the previous phase the vendor specific implementation is infused. Specifically, the workflow required by each provider is mapped to the abstract one defined for the particular service. Likewise, the provider specific web API is mapped to the reference one.

iii) Execution phase. During that phase, the Platform Service Execution Controller (Fig. 1) handles the execution of the workflow, while the API Client Generator produces the code for the web API invocation of the chosen platform service provider.

In order to illustrate how the framework can be used in a real case scenario, the cloud payment service is used as an example in the rest of the paper. The payment service enables a website or an application to accept online payments via electronic cards such as credit or debit cards. This platform service has been chosen because of its inherent relative complexity compared to other services such as e-mail or message queue service. The complexity lies in the fact that the purchase transaction requires more than one step to be completed and there is a significant heterogeneity among the available payment providers with respect to the involved steps.

Fig. 2 describes the steps involved in completing a payment transaction, while Fig. 3 shows the state chart of the cloud application throughout the transaction.

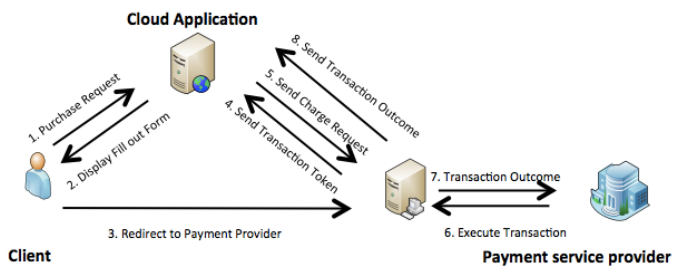
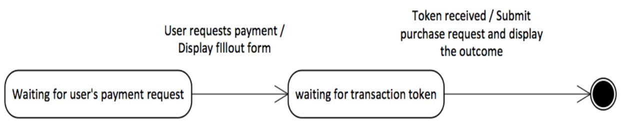


Figure 2. Cloud Payment Service



User requests payment / Display fillout form
Token received / Submit purchase request and display the outcome
Waiting for user's payment request
waiting for transaction token

Figure 3. State Chart of the Cloud Payment Service

Two states are observed. While the cloud application remains in the first state, it waits for a purchase request. Once the client requests a new purchase, the cloud application displays the fill out form where the user enters the payment details. Subsequently, the cloud application moves to the next state where it waits for the transaction token issued by the payment provider. The transaction token uniquely identifies the current transaction and can be used by the cloud application to complete the purchase. Once the user submits the form, she is redirected to the payment provider who validates the card details. Then a request to the cloud application is submitted including the transaction token. Once the token is received the application submits a request to the provider with the specific amount to be charged. The provider completes the transaction and responds with the outcome. Depending on the outcome, the cloud application displays a success or failure page to the client.

In the next section we describe in details the process of adding the payment service to the framework. As concrete payment provider, we use Spreedly an add-on offered via Heroku platform.

VII. Platform Service Workflow Modelling

A. *Platform Service Modelling Phase*

During this phase, the abstract functionality of the platform service is modelled. For that reason the reference meta-model shown in Fig. 4 is used. The meta-model comprises the following components, which enable the modelling of the workflow during the execution of an operation:

CloudAction: Cloud Actions are used to model stateful platform services, as described in Section II, which define more than one step in order to complete an operation. The whole process required to complete the operation can be modelled as a state machine. Each step can be modelled as a concrete state that the platform service can exist in. When the appropriate event arrives an action is triggered to handle the event and subsequently causes the transition to the next state. The events in this case are the incoming requests arriving either by the application user or the service provider. A separate Cloud Action is defined to handle each incoming request and subsequently signals the transition to the next state.

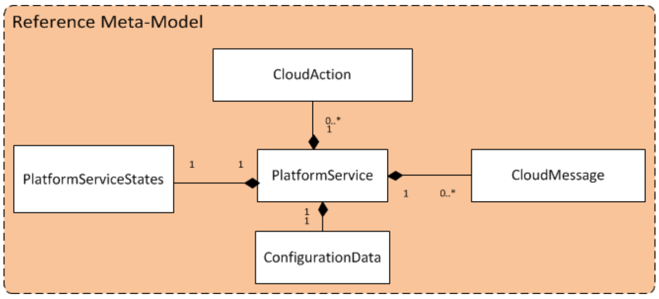


Figure 4. Reference Meta-model

*CloudMessage***.** CloudMessages can be used to model requests performed by the cloud application towards the service provider using the web API of the latter. The API usually conforms to the REST principles [19]. CloudMessages can either be used in stateless services, where the operation is completed in one step or within Cloud Actions when the latter are required to submit a request to the service provider.

*PlatformServiceStates***.** The PlatformServiceStates description file holds information about the states involved in an operation and the corresponding Cloud Actions which are initialised to execute the behaviour required in each state.

*ConfigurationData***.** Certain configuration settings are required by each platform service provider. That information is captured in the ConfigurationData. Example of settings which needs to be defined are the clients' credentials required to perform web requests and the redirect URL parameter which is often requested by the service provider in order to perform requests to the cloud application.

The reference meta-model is used to construct the Platform Service Constructor (PSC) as shown in Fig. 1. The PSC is a model of the abstract functionality of a given platform service. and is built based on the state chart defined for that service using the following rules:

1) For each state where the application waits for an incoming request, a CloudAction is defined to handle the request.

2) For each outgoing request to the service provider using the web API, a CloudMessage is defined.

In the case of the Cloud Payment Service, the middle component of the Fig. 5 shows the Cloud Payment Service Connector. It is constructed based on the state chart defined in Fig. 3 and using the reference meta-model. It consists of the following blocks:

FilloutForm. The FilloutForm is a CloudAction which receives the request for a new purchase transaction and responds to the client with the fill out form in order for the latter to enter the card details. The communication is realised using the servlet technology.

HandlePurchaseTransaction. The HandlePurchas Transaction is a CloudAction which receives the request from the service provider containing the transaction token. Then, a request is submitted to the provider including the transaction token and the amount to be charged. The provider replies with the outcome of the purchase and subsequently the action responds to the client with a success or fail message accordingly.

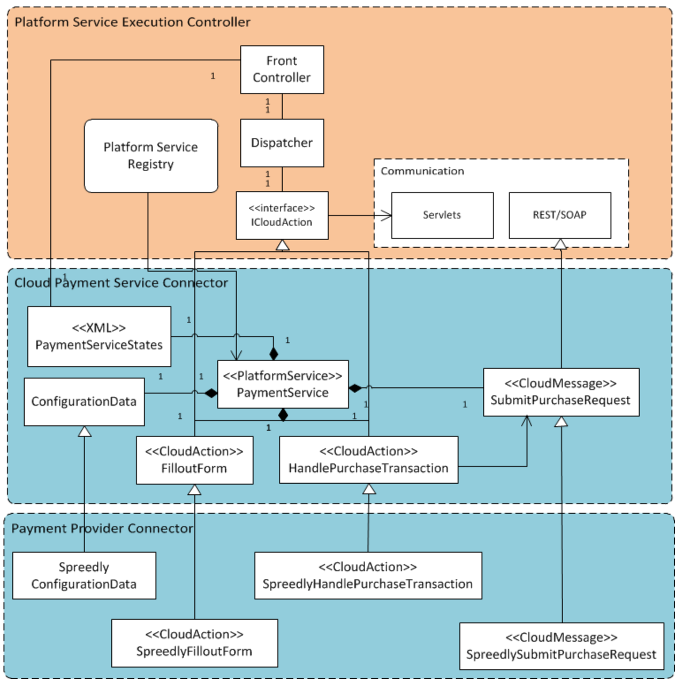
SubmitPurchaseRequest. The SubmitPurchaseRequest is a CloudMessage used internally by the HandlePurchase Transaction action. Its purpose is to model the request to the service provider, using the exposed web API, to complete the purchase transaction. It receives the provider's respond stating the outcome and forwards it to the action.

ConfigurationData. The ConfigurationData contains the service settings required to complete the purchase operation. Particularly, the following pieces of information are listed: the "redirectUrl", the username and the password.

PaymentSerivceStates. In the PaymentServiceStates file the states and the corresponding actions involved in the transaction are defined. The file is used by the framework to guide the execution of the actions. A part of the description file is shown here:

```
<StateMachine>
 <State name="PaymentForm"
   action="org.paymentservice.FillOutFormAction"
   nextState="SendTransaction"/>
 <State name="SendTransaction"
   action="org.paymentservice.SendTransactionAction"
   nextState="Finish"/>
</StateMachine>
```

At this point the Cloud Payment Service Connector (PSC) does not contain any provider specific information. Therefore any payment service provider which adheres to the specified model can be accommodated by the abstract model.



Platform Service Execution Controller
Front Controller
Platform Service Registry
Dispatcher
Communication
<<interface>> ICloudAction
Servlets
REST/SOAP
Cloud Payment Service Connector
<<XML>> PaymentServiceStates
ConfigurationData
<<PlatformService>> PaymentService
<<CloudMessage>> SubmitPurchaseRequest
<<CloudAction>> FilloutForm
<<CloudAction>> HandlePurchaseTransaction
Payment Provider Connector
Spreedly ConfigurationData
<<CloudAction>> SpreedlyHandlePurchaseTransaction
<<CloudAction>> SpreedlyFilloutForm
<<CloudMessage>> SpreedlySubmitPurchaseRequest

Figure 5. Cloud Payment Service Model

B. *Vendor Implementation Phase*

After having defined the PSC, the specific implementation and settings of each concrete providers needs to be infused. For each CloudAction and CloudMessage defined in the PSC, the respective provider specific blocks should be defined forming the Provider Connector (PC).

In the case of the payment service example, the Cloud Payment Provider Connector for the Spreedly provider is shown in the lower part of the Fig. 5. It contains the following blocks: (i) SpreedlyFilloutForm, (ii) SpreedlyHandlePurchaseTransaction and the (iii) SpreedlySubmitPurchaseRequest. In addition, the ConifgurationData file needs to be updated accordingly in order to match the specific provider.

Should the provider's implementation accurately matches the model, the provider specific Actions and Messages can reuse the functionality of the generic model. In case the provider's implementation diverts from the generic model the model's functionality can be overridden.

C. *Execution Phase*

During the execution phase the PSC and the PC, constructed in the previous phases, are managed by the Platform Service Execution Controller (PSEC) as shown in the Fig. 5. The PSEC automates the execution of the workflow required to complete an operation. It consists of the main following components shown in the upper part of the Fig. 5.

Front Controller. The Front Controller [20] serves as the entry point to the framework. It receives the incoming requests by the application user and the service provider.

Dispatcher. The dispatcher [21] follows the well-known request-dispatcher design pattern. It is responsible for receiving the incoming requests from the Front Controller and forwarding them to the appropriate handler, through the ICloudAction which is explained below. As mentioned in 3.1, the requests are handled by the CloudActions. Therefore the dispatcher forwards the request to the appropriate CloudAction. In order to do so, he gains access to the platform service states description file and based on the current state it triggers the corresponding action.

ICloudAction. ICloudAction is the interface which is present at the framework at design time and which the Dispatcher has knowledge about. Every CloudAction implements the ICloudAction. That facilitates the initialisation of the new CloudActions during run-time.

Communication patterns. Two types of communication pattern are supported by the framework: The first one is the Servlets and particularly the Http Servlet Request and Response objects [21] which are used by the Cloud Actions in order to handle incoming requests and respond back to the caller. The second type of communication is via the use of the REST/SOAP protocol which enable the CloudMessages to perform external requests to the service providers.

Cloud Service Registry. The Cloud Service Registry, as the name implies, keeps track of the services that the cloud application consumes.

In this section, we explained how the framework can be utilised in order to model the workflow of the platform services. The use of the reference meta-model enables the consistent modelling of the platform services while the construction of the Platform Service Connectors (PSC) allows the abstraction of the providers' peculiarities. The PSEC automates the execution of the workflow offloading the task from the application developer, to handle the various states.

VIII. Platform Service API Description

The second part in the process of adding a platform service and providers to the framework constitutes the description of the web API. As mentioned in Section II, the second variability point among platform services is the different web APIs that the concrete providers expose. Therefore, the heterogeneity of the web APIs shall be captured by the framework and abstracted by a common reference API exposed to the application developers.

In order to enable the uniform description of the platforms services' API, the benefits of ontologies are exploited. According to Gruber [22] ontologies are formal knowledge over a shared domain that is standardised or commonly accepted by certain group of people. The advantages here are two-fold. First, ontologies allow to define clearly the domain model of our interest; in our case the domain model is the platform service providers web API. The fact that an ontology can be a shared and a commonly accepted description of a platform service, contributes towards the homogenisation of the latter. The platform vendors can adhere to and publish the description of their service based on the common and shared ontology.

The reasoning capabilities that ontologies offer may be exploited for consistency check of the service descriptions and also for service discovery and recommendation.

Moreover, ontologies can be reused and expanded if necessary. Thus, an ontology describing a platform service may not be constructed from the ground up but may be based on an existing one. The intention of the authors is to reuse and expand the Linked USDL [23] ontology and particularly the extended Minimal Service Model (MSM) as described in [24]. To the best of our knowledge and according to [24] the MSM is the richest description model capable of capturing the web API and enabling automatic invocation.

The platform service API description is based on an hierarchy of a three level ontologies as shown in Fig. 6. Inspiration has been gained by the Meta-Object-Facility (MOF) standard [25] defined for the Model Driven Engineering domain. Specifically, the hierarchy of the ontologies resembles the bottom three levels of the MOF structure, namely the meta-models, the models and the instances of the models.

The level 2 Ontology (O2) includes the concepts required to describe a web API. Such concepts are the Operations offered by the service providers, the Parameters and the endpoint for each operation etc. The level 1 Ontologies (O1) include the concrete description of each of the platform services which are supported by the framework. A dedicated ontology corresponds to each of the platform services and captures information about the functionality that each of the services expose. For example, in the case of the cloud payment service, information related to charging or refunding a card is captured. The ontologies in the O2 level are also referred to as Template ontologies. The level 0 Ontologies (O0) include the description of the specific platform service providers. A dedicated ontology corresponds to each of the providers and describes the native web API. The ontologies in the O0 level are also referred to as Instance ontologies.

During the three phases we describe how the ontological service descriptions are formed and used to automatically generate the clients.

A. Platform Service Modelling Phase

During this phase, the platform service reference API, as shown in Fig. 1, is defined. The reference API is exposed to the application developers and describes the operations offered by the particular service. It is formed using the service API description editor which offers a user interface and is provided as plug-in in Eclipse IDE. The reference API is captured in the Template Ontology.

Fig. 7 shows a snapshot of the Template ontology for the payment service which describes the operation for charging a card. For the sake of simplicity only the necessary amount of information has been included. The name of the operation is "ChargeCard". It is a subclass of the class "Operation". "Operation" is defined in the Abstract platform service ontology (O2 level) and includes all the operations offered by the service. Fig. 7 also includes the following three elements: "CardIdentifier",which denotes the card to be charged, "ChargedAmount", which refers to the amount of money to be charged during the specific transaction and "CurrencyCode" which refers to the currency to be used for the specific transaction. All three elements are subclasses of the class "Attribute". The class "Attribute" is defined in the Abstract platform service ontology and includes all the attributes which are used for the execution of the operations. An attribute is linked to a specific operation with a property. Specifically, the three afore mentioned attributes are linked to the "ChargeCard" operation with the following properties respectively: "hasCardIdentifier", "hasChargedAmount", "hasCurrencyCode".

B. Vendor Implementation Phase

In this phase the provider specific web API is described and mapped to the reference API. The Service API description editor is used to perform the mapping. The outcome is an Instance ontology (O0 level) for each concrete provider.

Fig. 8 and 9 depicts two Instance Ontologies which correspond to two payment service providers offered by Heroku and Amazon respectively.

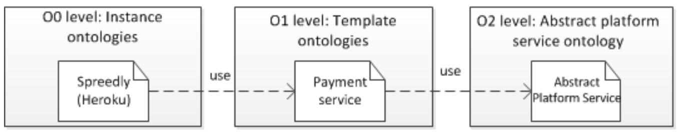

O0 level: Instance ontologies
Spreedly (Heroku)
use
O1 level: Template ontologies
Payment service
use
O2 level: Abstract platform service ontology
Abstract Platform Service

Figure 6. The three levels of the ontology hierarchy

Particularly, Fig. 8 shows the description of the charge operation as defined in the API of the Spreedly service offered via Heroku platform. Individuals are created to express each of the specific elements of the provider`s API. An Individual, in the field of Ontologies can be considered as an instance of a class. Specifically, the "purchase" Individual denotes the operation name which is equivalent to the "ChargeCard" operation of the Template Ontology. This justifies the fact that "purchase" individual is of type "ChargeCard". The Individual "amount" denotes the amount to be charged during the transaction and is equivalent to the "ChargedAmount" attribute. Thus it is defined of type "ChargedAmount". Likewise the Individual "currency_code" is of type "currency" and the "payment_method_token", which identifies the card to be charged, is of type "CardIdentifier".

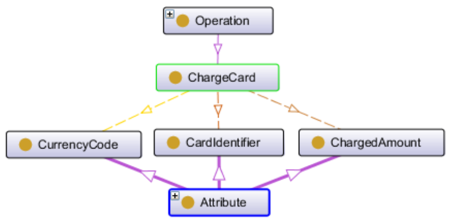


Figure 7. Example of Template ontology for the cloud payment service

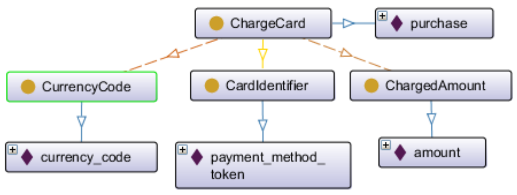


Figure 8. Example of Instance ontology for the Spreedly payment service

In the same way an Instance Ontology is created (Fig. 9) to describe the API of the "Stripe" payment service provider offered via Amazon. The individual "create" denotes the creation of a charge and is equivalent to the "ChargeCard". Therefore it is of type "ChargeCard". Likewise, the individuals "amount", "currency" and "card" are of type "ChargedAmount", "CurrencyCode" and "CardIdentifier" respectively.

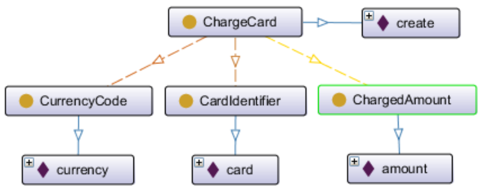


Fig. 9. Example of Instance ontology for the Stripe payment service

In the same way the rest of the functionality of a platform service can be described. At the same time, the differences in the APIs between the various providers can be captured. The payment service has been used as an example. The proposed structure of the three levels of ontologies can be used to describe the web API of additional platform services such as authentication and message queue service. Initially, a Template ontology is formed to describe the functionality of each of the platform services. Consequently the Instance ontologies are created to capture the vendor specific web APIs.

C. Execution Phase

During the Execution Phase, the Platform Service Reference and the provider specific API descriptions, which correspond to the Template and the Instance Ontologies respectively, are fed to the API Client Generator (Fig. 1). This component parses the Ontologies and generates:

(i) A set of interfaces which correspond to the reference API and provide the application developer with access to the functionally of the service.

(ii) The client code for the web API invocation of each of the concrete providers which implement the platform service.

Further information about the code generation can be found in [26]. Therefore the application developers can seamlessly deploy the platform service providers without being required to adhere to the specific web APIs or manually implement the client for each individual API.

IX. DISCUSSION AND CONCLUSIONS

This paper presented a development framework enabling the design of service-based cloud applications. Particularly, the framework facilitates the integration of platform basic services in a consistent way as well as seamless deployment of the concrete providers implementing those services. It achieves this by alleviating the variability issues that may arise across the platform services, namely: (i) the differences in the workflow when executing an operation, (ii) the heterogeneous web API exposed by the providers and (iii) the various configuration settings and authentication tokens that each provider requires. The main components of the framework are: (i) the reference meta-model, which enables the modelling of the abstract functionality of the platform basic service and an ontology-based architecture for alleviating the differences between the Providers' web APIs and automatically generating the client adapters for the API invocation. The process of adding a platform service provider in the framework is divided in three steps: (i) The Platform Service Modelling phase, where the abstract functionality is captured, (ii) the Vendor Implementation phase, where the specific provider functionality is infused and the (iii) Execution phase where the framework handles the operation execution.

The main limitation of the framework is that it is inherently restricted to the abstraction of the common features of the service providers. This means that the reference API contains the operations which are collectively offered by the supported providers. This is a natural limitation when dealing with API abstraction that is also encountered by similar solutions such as the jClouds, mOSAIC and TOSCA, which are involved with cloud services API abstractions. A solution to that is to provide the application developers with direct access to the client adapters for the specific provider when they need to use provider specific functionality, which is not addressed by the reference API. In addition, the reference API rather than being static, can be continuously updated to reflect the new features offered by the platform service providers.

The current version of the framework supports the abstraction of platform service providers and the management of the services that are integrated in the cloud application. Future work involves the expansion of the framework so that it offers functionality for automatic discovery and recommendation of services. Furthermore, it can provide billing information about the incurring costs of the application with respect to the services that it consumes.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nº264840, the RELATE project (http://www.relate-itn.eu).

REFERENCES

[1] F Gonidis, I. Paraskakis, A. J. H. Simons and D. Kourtesis, "Cloud Application Portability. An Initial View", in *6th Balkan Conference in Informatics*, Thessaloniki, 2013, pp. 275-282.

[2] D. Kourtesis, K. Bratanis, D. Bibikas, and I. Paraskakis, "Software Co-development in the Era of Cloud Application Platforms and Ecosystems: The Case of CAST", in *Collaborative Networks in the Internet of Services*, Bournemouth, 2012, pp. 196–204.

[3] J. Guillen, J. Miranda, J. M. Murillo and C. Cana, "Developing migratable multicloud applications based on MDE and adaptation techniques", in *the 2nd Nordic Symposium on Cloud Computing & Internet Technologies*, Oslo, 2013, pp. 30-37.

[4] F. Gonidis, "Experimentation and Categorisation of Cloud Application Platform Services", South East European Research Centre (SEERC), Thessaloniki, 2013.

[5] Heroku. (2014). [Online]. Available: http://heroku.com

[6] Google App Engine. (2014). [Online]. Available: https://developers.google.com/appengine.

[7] AWS Marketplace: Find and Buy Server Software and Services that Run on the AWS Cloud. (2014). [Online]. Available: https://aws.amazon.com/marketplace.

[8] S. Pautasso, O. Zimmerman andF. Leymann, "Restful web services vs. "big"' web services making the right architectural decision." In *17th International Conference on World Wide Web*, Beijing, 2008, pp. 805-814.

[9] jclouds. (2014). [Online]. Available: http://www.jclouds.org.

[10] Apache LibCloud. (2014) [Online]. Available: https ://libcloud. apache.org/index.html.

[11] D. Petcu, "Consuming Resources and Services from Multiple Clouds in Journal of Grid Computing", nr. 10723, Jan 2014, pp. 1-25.

[12] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann and A. Nowak, "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications" in 11th International Conference, ICSOC 2013, Berlin, 2013, pp. 692-695.

[13] T. Binz, G. Breiter, F. Leymann and T. Spatzier, "Portable Cloud Services Using TOSCA in IEEE Internet Computing" vol. 16, May-Jun 2012, pp. 80–85.

[14] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi and S. Mosser, "MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds" in *Workshop on Modeling in Software Engineering*, Zurich, 2012, pp. 50-56.

[15] K. Jeffery, G. Horn and L. Schubert, "A vision for better cloud applications" in Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds, Prague, 2013, pp. 7-12.

[16] A. Rossini, N. Nikolov, D. Romero, J. Domaschka, K. Kritikos, T. Kirkham and A. Solberg, "D2.1.2 - CloudML Implementation Documentation (First version)", CloudML, 2014.

[17] M. Hamdaqa, T. Livogiannis and L. Tahvildari, "A reference model for developing cloud applications" in 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, 2011, pp. 98-103.

[18] M. Maleshkova, C. Pedrinaci and J. Dominique, "Investigatingweb APIs on the World Wide Web", in 8th European Conference on Web Services (ECOWS), Ayina Napa, 2010, pp. 107-114.

[19] R. Fielding, "Architectural styles and the design of network-based software architectures", PhD thesis, University of California, Irvine, 2000.

[20] D. Alur, J. Crupi and D. Malks, "Core J2EE Patterns. Sun Microsystems Press", 2001.

[21] J. Hunter and W. Crawford, "Java Servlet Programming", O'Reilly & Associates, Inc., Sebastopol, CA, 2001.

[22] T. R. Gruber, "A translation approach to portable ontology specifications in Knowledge Acquisition", vol. 5, Jun. 1993, pp. 199–220.

[23] C. Pedrinaci, J. Cardoso, and T. Leidig, "Linked USDL: A Vocabulary for Web-scale Service Trading", in 11th Extended Semantic Web Conference (ESWC 2014), Crete, 2014.

[24] L. Ning, C. Pedrinaci, M. Maleshkova, J. Kopecky and J. Domingue, "OmniVoke: A Framework for Automating the Invocation of Web APIs" in *Fifth IEEE International Conference on Semantic Computing*, Palo Alto, 2011, pp. 39-46.

[25] T. Gardner, C. Griffin, J. Koehler and Rainer Hauser, "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard", in *Workshop on Metamodeling for MDA*, York, 2003, pp. 179–197.

[26] F. Gonidis, I. Paraskakis and A. J. Simons, "On the role of ontologies in the design of service based cloud applications", in *Second Workshop on Dependability and Interoperability in Heterogeneous Clouds,* Porto, 2014, in press.