Computers and Electrical Engineering 56 (2016) 842–853

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Pattern-based orchestration and automatic verification of composite cloud services

Flora Amato[a,*], Francesco Moscato[b]

[a] DIETI, University of Naples Federico II, Italy
[b] DiSciPol, Second University of Naples, Italy

ARTICLE INFO

Article history:
Received 29 August 2015
Revised 7 August 2016
Accepted 8 August 2016
Available online 24 August 2016

Keywords:
Cloud patterns
Verification
Semantics
Orchestration
Service level agreeement
Quality of service

ABSTRACT

Recent years have seen an increase of complexity in paradigms and languages for development of Cloud Systems. The need to build value added services and resources promoted pattern-based composition and orchestration as new hot research topics. Anyway, unlike web services, it is unclear what orchestration means for Cloud Systems. In this scenario, a way to automatically build composite services from their pattern-based description is appealing. In this work we describe a methodology for automatic composition and verification of Cloud Services which is driven by formal orchestration language.

1. Introduction

Cloud Architecture is nowadays a *de facto* standard for providing *any* kind of service on Internet. Big vendors, like Amazon Web Services (AWS)[1] or Microsoft with Azure[2] are definitely the main actors in Cloud Architecture definition. In [1] NIST extends the meaning of orchestration to Cloud Architecture. In addition, a new trend in Cloud Services design and management grew up in the last years: the definition of design patterns for Cloud Computing. Anyway, the introduction of design patterns in Cloud Computing requires a concept of orchestration that is more complex than the one defined in [1]: as we will show in this work many design patterns [2] with different purposes can be described as complex workflows. Workflow based composition opens the problem of understanding if a composite service is compliant with users' requirements and QoS. Properties of composite services obviously depend on components properties and composition. We think that compliance cannot be evaluated only by means of syntactical or type checking. Actually, several semantics-based approaches for *simple* web services composition exist (a survey is in [3]). Some of them exploit BPEL4WS [4] orchestration language and OWL-based ontologies for services description. In this context, it is clear that a methodology able to compose and verify Cloud Services by using Cloud Design Patterns is really appealing. This is the reason for we propose a formal orchestration language able to describe all elements, events and composition issues we need to describe complex services. The language enables pattern-based composition of Cloud elements at different layers. In this way, we reach two goals:

* Corresponding author.
E-mail addresses: flora.amato@unina.it (F. Amato), francesco.moscato@unina2.it (F. Moscato).

[1] https://aws.amazon.com
[2] https://azure.microsoft.com/

http://dx.doi.org/10.1016/j.compeleceng.2016.08.006
0045-7906/

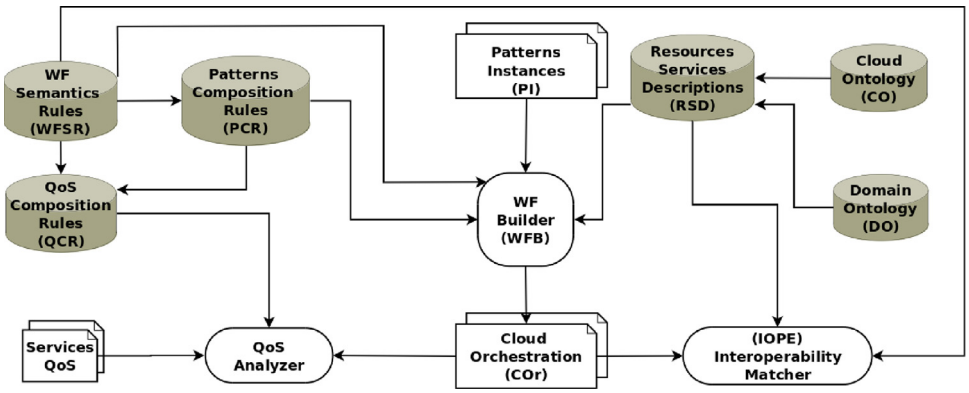


Fig. 1. System architecture.

first, our methodology enables Cloud designer and programmers to describe interactions of components directly by means of patterns; second, the proposed approach allows for the analysis of semantics and Quality of Services(QoS) of composite services and resources.

The paper is organized as follows. Section 2 contains the description of the methodology we propose and the architecture of a framework used to enact its steps. In order to describe patterns and composite services in Section 3 we introduce an orchestration language. Section 4 discusses the problem of binding real services into composite services. Section 5 proposes a full example and Section 6 contains an analysis of related works. Finally, Section 7 reports some concluding remarks.

2. Methodology and architecture

In the methodology we are going to illustrate, a formal workflow language describes composition and patterns. Its formal semantics, together with a semantic-based definition of cloud resources and services, enables automatic composition. Patterns give information about *how* services and resources interact at early design and development stages. This abstract information suggests the way to analyse, for example, composition soundness or quality of services. Let us consider a simple example with two basic control flow patterns: the *sequence* and the *1 out of N*. From an availability point of view, if N services are organized in a sequence, the failure probability of the composite service is the sum of the failure probabilities of components, on the other hand in the 1 out of N composition, it is their product. Our workflow language (Operational Flow Language: OFL in the following) has simple constructs that can be used to define complex Cloud Patterns[3] [2]. OFL is expressive enough to describe several patterns, as well as simple enough to be defined by a clear operational semantics. The final step is the management of Cloud composite services as patterns instances.

In order to bind real services in composition skeleton, we have to know their functionalities and their QoS properties. Our methodology supports ontology-based description of services by using OWL-S [5] but it is not enough expressive to characterize general composition [4]. We think that an Inputs, Outputs, Preconditions, Effects (IOPE [6]) characterization of Cloud resources, as well as a semantic description of what resources are and what they do, are useful to choose components that best match semantics and requirements of composite services. For proper matching and ontology [7] describes roles of available resources and services. Finally, for what QoS analysis and composition concerns, the workflow graphs described by OFL allow for the creation of analysis models by using Model Transformation techniques [8].

Fig. 1 depicts the architecture that enforces the methodology previously introduced.

The **WF Builder**, the **IOPE Interoperability Matcher** and the **QoS Builder** respectively manage goals of: (a) creating a workflow of composite services from their patter-based description; (b) matching component services in the orchestrated service; (c) analysing QoS of resulting composite Cloud Services. The WF (Workflow) Builder is based on a Knowledge Base of logic rules containing (a) the Operational semantics of OFL (in the repository **WF Semantic Rules**); (b) the operational semantics rules of orchestration patterns (**Patterns Composition Rules**). They are in turn defined by OFL. Results from WF Builder are skeletons used to implement Orchestrated Cloud Services. Proper back-end units can read skeletons in order to create stubs for different Vendors languages and APIs. Inputs for the Service Interoperability Matching are **Cloud Orchestration** skeletons (declaring how services interact) and the descriptions of component services and resources. Descriptions use OWL-S and IOPE grounding for cloud resources. In addition, the description of services functionalities depends on domain ontologies (i.e. financial services, E-Health etc.). Furthermore Cloud Ontology [7] describes the roles of elements in the Cloud Architecture. The description of these ontologies is out of the scope of this work. Finally, QoS Analyzer takes Cloud Orchestration and QoS descriptions in order to build analysable models (by using **QoS Composition rules**). At the moment the analyzer supports only performance and availability analyses.

[3] https://cloudpatterns.org, http://en.clouddesignpattern.org/index.php/Main_Page, https://msdn.microsoft.com/en-us/library/dn568099.aspx

Table 1
OFL main elements.

Element	SubElement	Description
Transition	ControlFlow	Defines precedences among activities in OFL control flows
	DataFlow	Declares data routing
	Event	Declares asynchronous events like interrupts
	DependsOn	Defines if an element in OFL depends on the existence or the correct execution of another element in process definition
	Handling	Declares relationships among Events and their managing activities
	InstalledOn	Defines where activity-related services are installed
	PoolCreate	declares that the source activity of this transition may create a persistent instance of a Pool
	PoolCreateOn	Like PoolCreate but it creates services on existing Virtual or physical resources
	PoolCreate1S	Creates a service and destroys when it ends
	PoolDestroy	Deallocate resources in a Pool
	Monitoring	Monitors resource
	PoolInvokeOn	Invoke a service in a Pool
Participant	Actor	Represents an external user for the process: it can send events and/or provide data
	PhysicalP	A physical resource
	VirtualP	A Virtual Server
Activity	Interruptible	Defines interruptible activities: these activities can be target of Event Transitions
	Handler	Handles events. Connected to an event by an Handling Transition
	NoReturnH	This is a Handler that does not return control to the interrupted activity
	ReturnH	This is a Handler that returns control to the interrupted activity
	Resource	A Cloud Resource that can execute some operation (for example a storage element)
	AbstractRes	Defines that the activity is an abstract resource
	PhysicalRes	Defines that the activity is at physical resource
	SaaS	Defines that the activity is at SaaS layer
	PaaS	Defines that the activity is at PaaS layer
	IaaS	Defines that the activity is at IaaS layer
	Monitor	Defines a monitoring activity
	Route	processes only inner data
Process Data	Data	Simple data like files, streams etc.
	Message	Data containing messages from an activity to another
Inner Data	Variable	A process variable
	Transition Condition	predicates that enables firing of transition
	LoopCond	Conditions for Loop blocks
	PoolCond	Conditions for Pool instantiation and activation
Block	Structural	a simple activities grouping
	Loop	Activities in a loop
	SubProcess	Declares sub-processes structures
Pool	A Pool is a collection of instances of activities that can be created dynamically. Once a Pool instance is created (by proper transition), it is associated to proper pool condition. Whenever an activity fires to a pool, the firing condition selects the proper instance of resource and services to use	

3. OFL semantics

In this section we briefly introduce the basic elements and operational semantics of OFL. Then, we show how OFL is expressive enough to describe Cloud Patterns. OFL is a workflow-based language: it consists of a graph of activities, participants and their properties. An Activity represents one logical step within a composite process. The completion of an activity and the starting of another one is a transition point in the workflow execution. Transitions may be unconditional, but the order of execution of activities at run-time may depend on one or more logical expressions called transition conditions. Conditions are evaluated when activities start and end and their values affect the behaviours of activities. More in detail, Table 1 lists the main constructs of OFL and their meanings.

Dependencies exist among elements in different groups. For example, Control Flow transitions can be connected only to activities; transition conditions exist only in Control Flow etc. We omit here all dependences for brevity. Some points exists within the workflow that allow for the control of activities execution: AND-split is a point where a single thread of control splits into two or more threads which are executed in parallel. AND-join is a point where two or more parallel activities converge into a single thread of execution. XOR-split is a decision point where only one of alternative branches is executed. XOR-join is a point in the workflow where two or more parallel activities converge in a single thread of execution without synchronization. OR-split is a decision point between several alternative workflow branches. OR-join is a point in which several alternative branches re-converge into a single thread. In the following XOR, AND and OR are called Split or Join Conditions. Therefore in the OFL language, workflow processes consist of a graph of nodes (activities) and edges (transitions) identified by a pair of nodes (*From, To*). Handling transitions are the only exception since they connect Event Transitions to Handler Activities. Activities represent atomic Cloud Service execution or resource uses. Types of elements

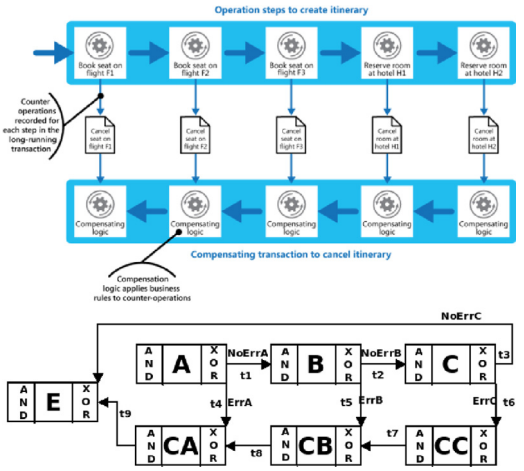


Fig. 2. OFG example.

in OFL graphs (Operational Flow Graph: OFG) are depicted as stereotypes (i.e. reported inside double-angled brackets). An example is in Fig. 2 that reports the OFG of an instance of the Compensating Transaction Pattern.

The workflow here is really simple: it includes only SaaS activities and ControlFlow transitions, hence, we omitted stereotypes for simplicity. Transitions conditions are reported near the related transitions. If they miss, we always evaluate conditions *true*. In the figure, **Err** and **NoErr** are true when an error or no errors respectively occurs during the execution of the *from* activity. Obviously they are mutually exclusive: an XOR split assures that only one between compensating and next activities are able to execute. If a the service executes a compensation activity, it executes the following compensations too.

3.1. Mapping Cloud Patterns to OFL graph

OFGs represent executable processes that can be easily implemented in different Cloud environments. We only need a way to generalize OFG in order to represent Cloud Patterns. Some patterns are not parametric in terms of component services and an OFG is enough to describe them. More generally, a Cloud Pattern is a parametric graph where services and resources involved in instances modify the pattern template. In order to allow for automatic composition, we must translate patterns into OFGs. The main structure of a pattern is a particular Block, called Template. We have defined an imperative language called *Pattern Grounding Language (PGL)* in order to describe Blocks replicas and groundings. We do not describe PGL here for brevity: its main features are the ability of defining sets (lists) of Services that are grounded into an OFG that describes patterns instances. A proper function, *connect*, links services and resources each other by means of proper Transitions. Let us consider the Compensation Pattern: a Designer would describe the pattern in the following way:

$$compensation\_pattern(S, Services, Compensating, E)$$

where *Services* is a list of normal services and *Compensating* is the list of services that compensate the ones in *Services*. *S* and *E* are the start and end activities respectively. The general compensation pattern is a Block coupled with a PGL definition. Fig. 3 reports the OFG and the PGL description for compensation pattern.

It is simple to prove that the PGL on the right of Fig. 3, invoked with parameters:

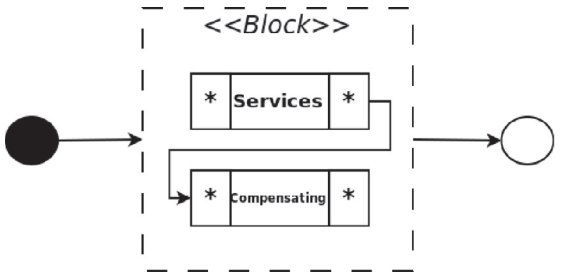
$$Compensation(S, [A, B, C], [CA, CB, CC], E)$$

produces the OFG in the bottom of Fig. 2.

4. Conditions propagation and semantics

In previous sections we outlined that we are interested in pattern-based composition and orchestration of Cloud service. Starting from an abstract pattern-based description we provide a sets of rules able to translate pattern-based description into a workflow process. Composition must be verified in order to understand if components *match* inputs, outputs and semantics required by other connected elements. Matching is part of verification process that should control if Inputs and Outputs are in the right format for connected elements and if semantics of Cloud Services and resources are correct. At this purpose, we inherit some Web Service languages and methods: the Semantic Web Community created OWL-S [9], and matching algorithms, methods and tools based on IOPE (Inputs, Outputs, Preconditions and Effects) analyses [10]. The matching process requires the existence of a Knowledge Base $\mathcal{KB}$ and of a set of Inference Rules $\mathcal{IR}$. A formal, explicit description of the Domain is given by OWL and OWL-S. With reference to Fig. 1,

$$\mathcal{KB} = \{RSD \cup CO \cup DO\}; \mathcal{IR} \subset WFSR$$



$Compensation(S, Services, Compensating, E) :$
$Prec(C) \quad =$
$size(Services) == size(Compensating)$

Condition C_{ok} = List[size(Services)]
Condition C_{err} = List[size(Compensating)]
foreach $(c_1 : C_{ok}, c_2 : C_{err})$:
    $c_1.value = true; c_2.value = true;$
$s_H = head(Services); s_T = tail(Services);$
$C = reverse(Compensating);$
foreach $(srv : Services; cmp : C; ok : C_{ok}, err : C_{err})$:
    $if(srv == s_H)$:
        connect(srv,cmp,ControlFlow,err);
        connect(S,srv,ControlFlow,"true");
    $else\ if(srv == s_T)$:
        connect(srv,E,ControlFlow,"true");
    connect(srv,succ(srv,Services),ControlFlow,ok);
    connect(srv,cmp,ControlFlow,err);
foreach $(srv : Services)$:
    $srv.JoinCond = AND; srv.SplitCond = XOR;$
$S.SpliCond = AND; E.JoinCond = XOR$

Fig. 3. Compensating pattern OFG and PGL.

Elements in $\mathcal{KB}$ are axioms derived from a Domain ontology. Relationships among data and operations depend on the semantic description of operations, Pre-conditions, Effects, Inputs and Outputs. Rules for reasoning on $\mathcal{KB}$ are defined in operational semantics. $\mathcal{IR}$ contains the rules to apply in order to build valid OG in terms of IOPEs. The inference rules do not depend on the Domain. Notice that matching depends only on OFG structure. Hence, a Cloud resource can be grounded to an activity if: (a) its Input formats (if any) meet Output formats of resources and services of incoming activities in the OFG; and if: (b) All of its Preconditions are satisfied. Input and Output matching is relatively simple to analyse as well as I/O problems are quite simple to manage: if the required input is not available, probably the whole process is not correct; if the required input exists but in different format, proper wrappers can be used in order to translate data formats.

Preconditions and Effects (PE) are more difficult to manage. They are defined by logic predicates and they express concepts present both in *Cloud Ontology(CO)* and *Domain Ontology (DO)* (see Fig. 1). After the execution of a Cloud Service, some things may happen or change: these are the Effects that the service produces. Execution of Cloud Services may change the truth value of some predicates: this means that the PE matching depends on OFG too: a running composite service may evaluate some predicates true or false after the execution of *any* activity in the OFG. *Meeting of preconditions is assured by services previously executed at any time in the composite process and condition values may change during its execution*.

This is an effect we call **Condition Propagation** and it provides a mean to describe the semantics of the whole composite service.

Solving the **matching problem** requires that a resource or service: (a) accepts all the inputs required by the IOPE specification of a node in the OFG, (b) produces all outputs required by the IOPE specification of a node in the OFG, (c) satisfies all preconditions required by the IOPE specification of a node in the OFG, (d) produces all effects required by the IOPE specification of a node in the OFG.

The real problem is to understand if a Composite service is sound in terms of IOPE matching. Checking of adjacent elements in an OFG is simple: Outputs of incoming nodes, must comply with Inputs needed by the node we are checking. The problem of PE matching is more complex: for each node in the graph, the matcher must analyse many predicates in the workflow graph. An effect enables new predicates or changes truth values of existing ones. In addition, and this is the case of our example, during the execution of a composite service, some predicates may change their unification with variables.

Preconditions of an activity depend on Effects of activities executed *before*. Anyway, the set of these activities is not uniquely determined: thanks to different Join and Split conditions instances of an OFG run across different paths of the graph (for example this happens at OR and XOR split points).

Let *Prec*(*N*) and *Eff*(*N*) be the sets of preconditions and effects respectively, for a node *N* in the OFG. Let us call:

$$eval(p) \longrightarrow \{true, false\}, p \in Prec(N) or Eff(N)$$

the function that evaluates a truth value of a precondition or an effect.

Let us consider the sets *PL* (predicates lists) composed by the couples (p^{node}, $eval(p^{node})$) where $p^{node} \in Prec(node) \cup Eff(node)$ and $eval(p^{node})$.

$$PL^{node} = \{\left(p_1^{node}, eval(p_1^{node})\right), \cdots, \left(p_n^{node}, eval(p_n^{node})\right)\}$$

We consider the union of preconditions and effects because, for Condition Propagation, we assume that, in order to bind a real service into the workflow, it must meet all preconditions and, after its execution, it must produce all declared effects. The important is to understand if a service can avoid meeting a precondition because of Condition Propagation. We call $\mathcal{PH}^{node}$ (Possibly Happens) the set with all possible configurations of predicates lists (i.e. predicates with their evaluations)

for a node in the OFG.

$$\mathcal{PH}^{node} = \{PL_1^{node}, \cdots, PL_n^{node}\}$$

Notice that a PL^{node} in $\mathcal{PH}^{node}$ contains all predicates (with their evaluation) evaluated before the execution of *node* activity (we call this set: $\mathcal{PH}^{node}_{before}$) as well as its *Effects*. These may eventually substitute the truth value of predicates previously evaluated in the case *Effects* contain predicates that have been already considered. If *Effects* contain new predicates, they are added to all *PLs* in $\mathcal{PH}$: if

$$\mathcal{PH}^{node}_{before} = \{PL^{node}_{before_1}, \cdots, PL^{node}_{before_k}\}$$

let $Eff_{common_i}(node)$ the set containing all predicates (and their evaluation) in $PL^{node}_{before_i} \in \mathcal{PH}^{node}_{before}$ that are in $Eff(node)$ even with same or different evaluation.

$$\mathcal{PH}^{node} = \{PL^{node}_{before_i} - Eff_{common_i}(node) \cup Eff(node)\}, i \in (1..k)$$

In brief, we add new effects with proper evaluation to each Predicate List and eventually we substitute old evaluations in the Effects list. Multiple *PLs* in $\mathcal{PH}$ set represent the fact that OFG may contain different paths from a start to end points. This obviously happens when the composition contains choices or conditional execution of paths, but having multiple *PLs* is useful also in parallel paths when they manages preconditions and effects of the same predicates. If OFG is a simple sequence of nodes, for each node, the cardinality of $\mathcal{PH}$ is 1 for each node in the sequence: $card\left(\mathcal{PH}^{node}\right) = 1\ \forall node \in OFG$. This because in a simple sequence of nodes there is always one possible evaluation of predicates. Anyway, for more complex OFGs, we have:

$$card\left(\mathcal{PH}^{node}\right) \geq 1\ \forall node \in OFG$$

Let us now describe how $\mathcal{PH}^{node}$ sets are built for all nodes in OFG. We visit all the OFG from the start to the END points updating *PL* sets: updates depend on Effects and Split and Join points in the OFG. The simplest case, as we introduced before, is the *pure* sequence path in OFG. If we have two nodes in a sequence (let us call them *A* and *B*, with *B* following *A* in the graph), with *A* as start point and *B* the end point (i.e. a graph with only two nodes), we have:

$$\mathcal{PH}^B = \{PL_i^B = \left(PL_i^A \cup (eval(Prec(B))\right) - Eff_{common}(B)$$
$$\cup Eff(B)\} \forall PL_i^A \in \mathcal{PH}^A$$

Notice that, if *A* is a *starting point* in the process:

$$\mathcal{PH}^A = \{eval(Eff(A) \cup Prec(A))\}$$

If the in place of A a more complex graph exists, we must distinguish two cases for *B* depending on its *join* condition. The first case is when the *join* condition of *B* is **OR** or **XOR**: Here, we can build the set $\mathcal{PH}^B_{before}$ and then the $\mathcal{PH}^B$ with the rule previously reported.

$$\mathcal{PH}^B_{before} = \bigcup_{t \in incoming(B)} \left(\mathcal{PH}^{from(t)} \cup eval(Prec(B))\right)$$

In XOR and OR join, simply all previous preconditions lists (in all incoming paths) are considered. Notice that if two preconditions lists exist with both same precondition and evaluation, their union results in an unique PL. The other case is when the *join* condition of *B* is **AND**. This is more difficult to manage because two further cases are possible: (1) the set of predicates that change from the split node of parallel incoming paths, during paths executions are disjoint on different paths; (2) some predicates are in the Effects sets of nodes belonging to different parallel paths.

Let *SPN* be the name of the (split) node where the parallel paths begin, and *JPN* the join point we are considering. Let us call

$$CH^{node} = \{predicates \in first(PL^{node}) :$$
$$PL^{node} \in \{\mathcal{PH}^{node} - \mathcal{PH}^{node}_{before}\}$$

the list of predicates that change their evaluation or that are inserted into predicate lists after the evaluation of the effects in a node (*first* function here select the first element in a couple). If we consider a generic *path* ϕ in the OFG terminating in *node*, we can define the CH^{node} set for a whole path:

$$CH^{node}_{\phi} = \bigcup_{node \in \phi} CH^{node}.$$

In the first case, we have:

$$\bigcap_{\phi \in \text{SPN incoming paths}} CH^{node}_{\phi} = \emptyset$$

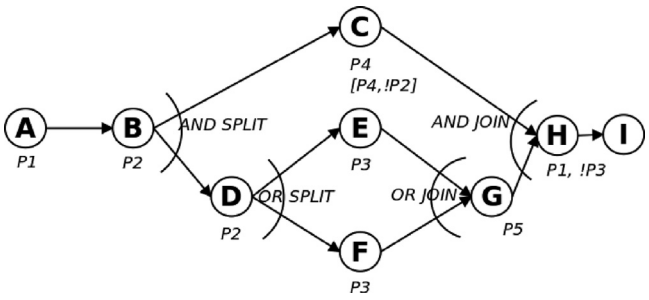


Fig. 4. PE matching example.

And

$$\mathcal{PH}^{JPN}_{before} = \{PL'_i = PL_i - \{(p_k, eval(p_k)) : p_k \in CH^{JPN}_{\phi}\} \cup \{(p_l, eval(p_l)) \in PL_m \in \mathcal{PH}^{I} : p_l \in CH^{JPN}_{\phi}\}\} \quad (1)$$

$\forall PL_i \in \mathcal{PH}^{SPN}, \forall \phi$ from SPN to JPN,

$\forall I \in from(incoming(JPN))$

This means that the $\mathcal{PH}^{JPN}$ is the same of the $\mathcal{PH}^{SPN}$ except for the predicates that have changed in all *JPN* incoming paths. In addition, predicates originally not in $\mathcal{PH}^{SPN}$ in parallel paths ϕ are added in $\mathcal{PH}^{JPN}$ as well.

The second case is more complicated: it involves the case when the same predicate changes in different parallel paths. It is usually a case of erroneous design (it figures like an anomaly in a transaction based system), but we consider it anyway since sometimes this behaviour may be explicitly wanted. Let us consider a partition Φ of all paths ϕ connecting *SPN* to *JPN*:

$$\Phi = \{\Phi_{NC} \cup \Phi_C\}, \Phi_{NC} \cap \Phi_C = \emptyset$$

where Φ_C is the set of paths where a predicate change exists. In the second case we have:

$$\mathcal{PH}^{JPN}_{before} = \mathcal{PH}^{JPN}_{before_{NC}} \cup \mathcal{PH}^{JPN}_{before_C} \quad (2)$$

$\mathcal{PH}^{JPN}_{before_{NC}}$ is computed as described in (1) but only on paths in Φ_{NC}. In addition,

$$\mathcal{PH}^{JPN}_{before_C} = \left(\bigcup_{Inc \in from(incoming(JPN)) \text{ on } \Phi_C} \mathcal{PH}^{Inc} \right) \cup eval(Prec(JPN))$$

We have now all elements to understand if an OFG is sound in terms of (IO)PE Matching. We analyse all nodes in the OFG, building for each node the sets $\mathcal{PH}^{node}$ and $\mathcal{PH}^{node}_{before}$. Then we analyse again the OFG. Two cases may happen:

$$\forall node \in OFG, Prec(node) \sqsubseteq PL_i \forall PL_i \in \mathcal{PH}^{node}_{before} \quad (3)$$

$$\forall node \in OFG, \exists PL_i \in \mathcal{PH}^{node}_{before} : Prec(node) \sqsubseteq PL_i \quad (4)$$
$$\text{AND } \exists j : PL_j \in \mathcal{PH}^{node}_{before} : \neg\left(Prec(node) \sqsubseteq PL_j\right)$$

In the case (3), all execution paths allow for the execution of a service in *node* if it meets the specified *Prec*(*node*) preconditions. The case (4) is more complex: at least a path exists where a component with *Prec*(*node*) preconditions can be executed without problem but at least one path exists too where a service meeting only preconditions in *Prec*(*node*) cannot be executed. In this case, the matching algorithms alerts users that something may go wrong during the execution of the composite cloud service. If the problem was an incorrect design, users can solve the problem by analysing $\mathcal{PH}^{node}_{before}$. Otherwise, proper compensation elements can be introduced in order to correct problem at run-time.

In order to provide an example of how $\mathcal{PH}$ sets are built, let us consider Fig. 4. It depicts an OFG where bold letters represent nodes (activities) names. Predicates appear italicized below each node. Without losing generality we report here only effects. The presence of the name of a predicate, indicates that the process evaluates it *true* after the execution of the related activity, while the presence of an exclamation mark means that the process evaluates the predicate *false*. Split and Join conditions are reported too. The figure shows two cases: in the first case, the $LP^C = \{(P_4, true)\}$, in the second case, $LP^C = \{(P_4, true), (P_2, false)\}$.

In the first case (Fig. 5 on the left), if $Prec(I) = \{Prec_1, Prec_2, P_1, P_4\}$ we can bind to *I* a service with precondition $Prec(I) = \{Prec_1, Prec_2\}$ since P_1 and P_4 are propagated during process execution. The same service cannot bind to *I* in the second case (Fig. 5 on the right), because there is a case where $\mathcal{PH}^H$ that does not contain P_4 and we are in the case of (4). Notice that $\mathcal{PH}^H$ describes the semantics of all the composite process except for the execution of *I*.

First Case: $Eff(C) = \{(P_4, true)\}$

A:
$\mathcal{PH}^A = [((P_1, true))]$
B:
$\mathcal{PH}^B_{before} = [((P_1, true))]$
$\mathcal{PH}^B = [((P_1, true), (P_2, true))]$
C:
$\mathcal{PH}^C_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^C = [((P_1, true), (P_2, true), (P4, true))]$
D:
$\mathcal{PH}^D_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^D = [((P_1, true), (P_2, true))]$
E:
$\mathcal{PH}^E_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^E = [((P_1, true), (P_2, true), (P_3, true))]$
F:
$\mathcal{PH}^F_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^F = [((P_1, true), (P_2, true), (P_3, true))]$
G:
$\mathcal{PH}^G_{before} = \mathcal{PH}^E_{before} \cup \mathcal{PH}^F_{before} =$
$[((P_1, true), (P_2, true), (P_3, true)),$
$((P_1, true), (P_2, true), (P_3, true))]$
Collapsing Union:
$\mathcal{PH}^G_{before} = [((P_1, true), (P_2, true), (P_3, true))]$
$\mathcal{PH}^G = [((P_1, true), (P_2, true), (P_3, true), (P_5, true))]$
H:
$Change_\phi = \emptyset$ for all paths from B to H
$\mathcal{PH}^H_{before} = [((P_1, true), (P_2, true), (P_3, true), (P_5, true), (P_4, true))]$
$\mathcal{PH}^H = [((P_1, true), (P_2, true), (P_3, false), (P_5, true), (P_4, true))]$

Second Case: $Eff(C) = \{(P_2, false), (P_4, true)\}$

A:
$\mathcal{PH}^A = [((P_1, true))]$
B:
$\mathcal{PH}^B_{before} = [((P_1, true))]$
$\mathcal{PH}^B = [((P_1, true), (P_2, true))]$
C:
$\mathcal{PH}^C_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^C = [((P_1, true), (P_2, false), (P4, true))]$
D:
$\mathcal{PH}^D_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^D = [((P_1, true), (P_2, true))]$
E:
$\mathcal{PH}^E_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^E = [((P_1, true), (P_2, true), (P_3, true))]$
F:
$\mathcal{PH}^F_{before} = [((P_1, true), (P_2, true))]$
$\mathcal{PH}^F = [((P_1, true), (P_2, true), (P_3, true))]$
G:
$\mathcal{PH}^G_{before} = \mathcal{PH}^E_{before} \cup \mathcal{PH}^F_{before} =$
$[((P_1, true), (P_2, true), (P_3, true)),$
$((P_1, true), (P_2, true), (P_3, true))]$
Collapsing Union:
$\mathcal{PH}^G_{before} = [((P_1, true), (P_2, true), (P_3, true))]$
$\mathcal{PH}^G = [((P_1, true), (P_2, true), (P_3, true), (P_5, true))]$
H:
$Change_\phi = P_2$ for all paths from B to H:
$\Phi_C = \{\{C\}, \{D, E, G\}, \{D, F, G\}\}$
$\mathcal{PH}^H_{before} = [((P_1, true), (P_2, false), (P_4, true)),$
$((P_1, true), (P_2, true), (P_3, true), (P_5, true)),$
$((P_1, true), (P_2, true), (P_3, true), (P_5, true))]$
Collapsing Union:
$\mathcal{PH}^H_{before} = [((P_1, true), (P_2, false), (P_4, true)),$
$((P_1, true), (P_2, true), (P_3, true), (P_5, true))]$
$\mathcal{PH}^H = [((P_1, true), (P_2, false), (P_3, false), (P_4, true)),$
$((P_1, true), (P_2, true), (P_3, false), (P_5, true))]$

Fig. 5. Propagation.

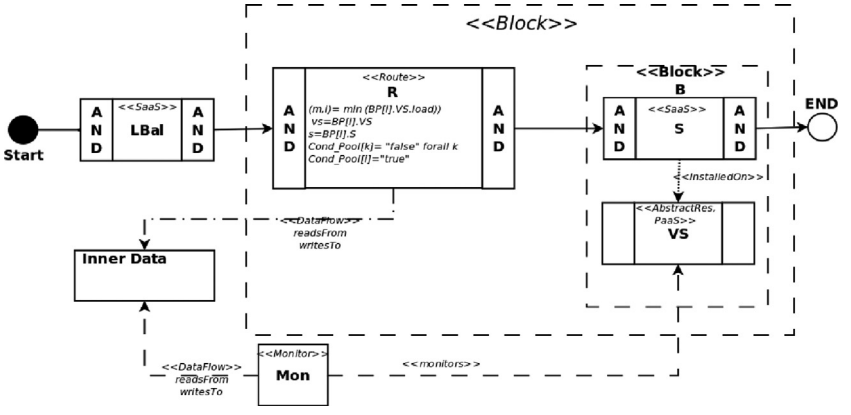


Fig. 6. Block OFG for Load Balancer.

5. Load Balancer full example

In this section we describe all steps needed to provide an OFG description of the Load Balancer Pattern.[4] Then we will discuss how we study matching on the realized pattern. The steps we discuss are the following: (1) we define a parametric OFG for the pattern and the related PGL; (2) we analyze soundness for a given instance of the pattern; (3) once some IOPE specification for the pattern are reported on the OFG, we show how matching is enacted.

Fig. 6 shows the parametric OFG for the Load Balancer Pattern. In the figure, *LBal* is the Load Balancer service which interfaces with monitored services; *S* and *VS* are respectively the generic Service and Virtual Server where the service is installed. They are collected in a Block. The Route Activity choose the Service in the Pool for load balancing purposes. It actually selects the virtual server with minimal load. This updates a list of transition conditions (*Cond_Pool*): the only

[4] http://cloudpatterns.org/design_patterns/service_load_balancing

$$LoadBalancer(S, Service, VirtualServer, instances, E) :$$
$$Prec(LoadBalancer) = (size(Services) == size(VirtualServers)), type(Service) == SaaS, type(VirtualServer) == PaaS$$

Saas LBal = SaaS();
Condition C_{bal} = List[instances];
Route r = Route("lbal.alg");
Monitor m = Monitor();
SaaSActivity Services = List[instances];
PaaSActivity VirtualServers = List[instances];
VirtualServerData[] VSD= List[instances];
createInnerDataVector(VSD);
ServiceData[] SD= List[instances];
createInnerDataVector(SD);
connect(lbal,r,ControlFlow,true);
foreach $(s : Services, vs : VirtualServers, c : C_{bal}, i = (1 \cdots instances))$:
bind(Service,s);
bind(VirtualServer,v);
connect (s,vs,installedOn,0);
connect (r,s,ControlFlow,c);
connect (s,E,ControlFlow,true);
c.value = "false";
connectMonitor(m,vs,monitors,VSD,i);
$S.SpliCond = AND; E.JoinCond = XOR$

Fig. 7. Load Balancer PGL.

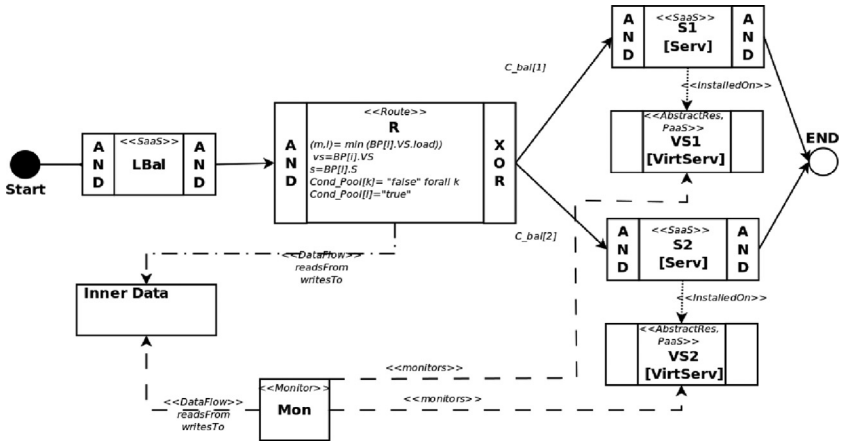


Fig. 8. Load Balancer workflow.

condition true is the one that will connect the route activity *R* with chosen service in the Block. The Monitor activity Mon stores load and performances measures in the Inner Data repository of the Process.

Fig. 7 reports PGL definition for the Load Balancer; data flow is omitted for brevity's sake. New elements to introduce are: the instructions to create data into Inner Data Section (i.e. VirtualServerData and ServiceData, that contains all information needed to record monitored values); the bind function that link a SaaS or a PaaS to the related activity. Notice that *lbal.alg* contains the text that appears in the route activity box. If we instances this pattern with:

$$LoadBalancer(S, Serv, VirtServ, 2, End)$$

the generated workflow is the one depicted in Fig. 8.

This is the point where we define the semantics of the composite service. Composite service description is defined here in terms of structure (patterns and OFL). We can use Preconditions and Effects definitions for adding semantics descriptions. The goal here is to define a Load Balanced GPS navigator service. Hence, load balanced services must compute a path between two GPS coordinates. The OFG we use for semantics analysis is in Fig. 9 that reports basic (IO)PE information near each node.

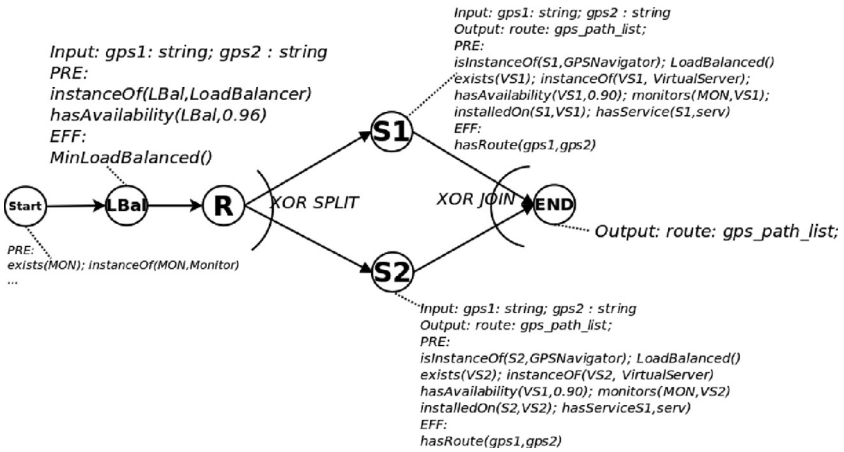


Fig. 9. OFG with IOPE for Load Balancer.

It is easy to prove that the precondition *LoadBalanced* on both S1 and S2 is satisfied for any service installed since it is propagated from $\mathcal{PH}^{S1}_{before}$ and $\mathcal{PH}^{S2}_{before}$: bounded services have to verify only the other preconditions. We can use as S1 service, any Cloud Service meeting all S1 precondition except *LoadBalanced*: the S1 service has to be a *GPSNavigator* service, that computes a route from *gps*1 to *gps*2 coordinates; S1 must be installed on a monitored Virtual Server with a given availability. If all precondition are satisfied in the OFG, The End Activity has the *hasRoute* effects from *gps*1 and *gps*2 coordinates, all preconditions in Start, S1 and S2 nodes, as well as *minLoadBalanced*. Information from transitions in Fig. 8 that is not related to control flow (i.e. all *dotted* transitions) generates Precondition for the Start Activity, like *installedOn* or *monitors* predicates.

6. Related works

In the early 2000s service composition was introduced and investigated for web services [11,12]. In particular, BPEL4WS [13] was elected by W3C consortium as reference language for composition of Web Services. NIST definitions and standards for Cloud Computing included Composition of Services by means of Orchestration only in the last years [1] but they are still far from formalizing composition and orchestration in the same way it was done for web services. The main effort in composition during last years focused on the choice of services and resources to use in a composite Cloud Service in order to improve Quality of Service [14]. Many works deal with optimization problems [15–23]. Anyway, the most of these works lacks in a formal definition of the Orchestration problem. A tentative of providing a Cloud Orchestration Engine with an orchestration language is in [24] where COPE (Cloud Orchestration Policy Engine) is presented. Peer-to-peer and collaborative approaches to composition have been proposed too: in [25–27] authors show the usefulness of the platform not only for efficient and reliable distributed computing but also for collaborative activities and ubiquitous computing [28]. The only mature work on Orchestration was made by OASIS in the Topology and Orchestration Specification for Cloud Application (TOSCA) [29]. Several works have been reported in literature about Cloud Pattern exploitation [30], but in general they contain only descriptions of different design patterns. At the best of our knowledge, this is the only work addressing composition in terms of both Cloud Computing Patterns and Workflow patterns. We think that these two concepts are strictly related at different layers of abstraction. In addition, we address matching problem not as optimization problem. We provide a way to cope matching and analyses of the orchestration process. We also provide a formal language to describe composition in terms of workflow. The language is a workflow language, so it is different from the other declarative and imperative-scripting language usually used in literature.

7. Conclusions and future works

This paper presents a methodology able to build and analyse composite Cloud Services. The methodology is based on Cloud Patterns and Orchestration in terms of workflow activities. We enable Cloud Designers to specify patterns they need in composite service creation. Pattern-based specification is then automatically translated into a workflow process. We provide a formal language (OFL) for process description that takes into account of resources and services relationships at different Cloud Service Layers. OFL is formally defined and its operational semantics can be used to prove soundness of composite services. In addition we solve the problem of semantics-based matching and analysis of composite process by means of Condition Propagation. Future works include the definition of model transformation algorithms for the study of other properties like performances, deadlocks, availability etc.

References

[1] VV.AA. Us government cloud computing technology roadmap release 1.0 (draft). In: Special publication 500-293, 2. NIST; 2011. p. 1–85.
[2] Fehling C, Leymann F, Rütschlin J, Schumm D. Pattern-based development and management of cloud applications. Future Internet 2012;4(1):110–41.
[3] Dustdar S, Schreiner W. A survey on web services composition. Int JWebGrid Serv 2005;1(1):1–30.
[4] Lorenzo GD, Mazzocca N, Moscato F, Vittorini V. Towards semantics driven generation of executable web services compositions. Int J Softw JSW 2007;2(5):1–15.
[5] Martin D, Burstein M, Mcdermott D, Mcilraith S, Paolucci M, Sycara K, et al. Bringing semantics to web services with owl-s. World Wide Web 2007a;10(3):243–77.
[6] Medjahed B, Malik Z, Benbernou S. On the composability of semantic web services. In: Web services foundations. Springer; 2014. p. 137–60.
[7] Moscato F, Aversa R, Martino BD, Fortis T-F, Munteanu VI. An analysis of mosaic ontology for cloud resources annotation. In: IEEE proceedings of FedCSIS 2011 conference; 2011. p. 973–80.
[8] Moscato F, Amato F, Amato A, Aversa R. Model–driven engineering of cloud components in metamorp (h) osy. Int J Grid Util Comput 2014;5(2):107–22.
[9] Martin D, Burstein M, Mcdermott D, Mcilraith S, Paolucci M, Sycara K, et al. Bringing semantics to web services with owl-s. World Wide Web 2007b;10(3):243–77.
[10] Klein M, Konig-Ries B, Mussig M. What is needed for semantic service descriptions? a proposal for suitable language constructs. Int J Web Grid Serv 2005;1(3–4):328–64.
[11] Rao J, Su X. A survey of automated web service composition methods. In: Semantic web services and web process composition. Springer; 2005. p. 43–54.
[12] Milanovic N, Malek M. Current solutions for web service composition. IEEE Internet Comput 2004:51–9.
[13] Weerawarana S, Curbera F, Leymann F, Storey T, Ferguson DF. Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more. Prentice Hall PTR; 2005.
[14] Jula A, Sundararajan E, Othman Z. Cloud computing service composition: a systematic literature review. Expert Syst Appl 2014;41(8):3809–24.
[15] Gutierrez-Garcia JO, Sim KM. Agent-based cloud service composition. ApplIntell 2013;38(3):436–64.
[16] Liu Y, Li M, Wang Q. A novel user-preference-driven service selection strategy in cloud computing. Int J Adv Comput Technol 2012;4(21).
[17] Marrone S, Nardone R. Automatic resource allocation for high availability cloud services. Procedia Comput Sci 2015;52(1):980–7.
[18] Pllana S, Benkner S, Xhafa F, Barolli L. A novel approach for hybrid performance modelling and prediction of large-scale computing systems. Int J Grid Util Comput 2009;1(4):316–27.
[19] Erfani S, Malek M, Sachar H. An expert system-based approach to capacity allocation in a multiservice application environment. IEEE Netw 1991;5(3):7–12.
[20] Feng G, Buyya R. Maximum revenue-oriented resource allocation in cloud. Int J Grid Util Comput 2016;7(1):12–21.
[21] Ezugwu AE, Buhari SM, Junaidu SB. Resource management system for scientific virtual laboratory applications. Int J Grid Util Comput 2014;6(1):8–20.
[22] Mondol MAS, Akbar MM. A new approach to schedule workflow applications for advance reservation of resources in grid. Int J Grid Util Comput 2014;5(3):165–82.
[23] Flammini F., Marrone S., Mazzocca N., Pappalardo A., Pragliola C., Vittorini V. Trustworthiness evaluation of multi-sensor situation recognition in transit surveillance scenarios. Lec. Notes in C. Sc. 8128 LNCS2013:442–456.
[24] Liu C, Loo BT, Mao Y. Declarative automated cloud resource orchestration. In: Proceedings of the 2nd ACM symposium on cloud computing. ACM; 2011. p. 26.
[25] Barolli L, Xhafa F. Jxta-overlay: a p2p platform for distributed, collaborative, and ubiquitous computing. Ind Electron IEEE Trans 2011;58(6):2163–72.
[26] Xhafa F, Fernandez R, Daradoumis T, Barolli L, Caballé S. Improvement of jxta protocols for supporting reliable distributed applications in p2p systems. In: Network-based information systems. Springer; 2007. p. 345–54.
[27] French T, Bessis N, Xhafa F, Maple C. Towards a corporate governance trust agent scoring model for collaborative virtual organisations. Int J Grid Util Comput 2011;2(2):98–108.
[28] Barolli L, Xhafa F, Durresi A, De Marco G. M3ps: a jxta-based multi-platform p2p system and its web application tools. Int J Web Inf Syst 2007;2(3/4):187–96.
[29] Binz T, Breitenbücher U, Kopp O, Leymann F. Tosca: portable automated deployment and management of cloud applications. In: Advanced web services. Springer; 2014. p. 527–49.
[30] Leymann C.F.F., Retter R., Schupeck W., Arbitter P. Cloud computing patterns. 2014.

Flora Amato, Ph.D., is Assistant Professor at the Department of Electrical Engineering and Information Technology of University of Napoli Federico II. Her research activities mainly concern Formal Modeling , Verification Techniques, Knowledge Management, Information Extraction and Integration. She is author of more of 70 research papers, published on International Journals and Conference Proceedings, and she worked as Program Chair in several international Workshops.

Francesco Moscato, Ph.D., is Researcher and Assistant Professor Professor at Second University of Naples. His research activities include Formal Modeling, Model Driven Engineering and Verification, Intelligent Systems and Cloud Computing. He is author of many research papers, published on International Journals and Conference Proceedings, and he worked in Program Committees of several international Workshops.