Noname manuscript No.
(will be inserted by the editor)

Self-Configuring Cloud Application Mashup with Goals and Capabilities

Luca Sabatucci · Salvatore Lopes · Massimo Cossentino

Received: date / Accepted: date

Abstract Cloud mashup is a technique for the seamless composition of SaaS applications from several sources into a single integrated solution. This paper presents a general approach for automatically composing applications and services deployed over the Cloud. The proposed approach implies to encapsulate distributed processes into smart and autonomic entities, namely cloud capabilities. Despite the lack of a central mashup server, these processes are able to autonomously organize in order to establish different ways to address the desired result. The approach uses a couple of languages for describing respectively the mashup logic in terms of goals and the available functionalities in terms of capabilities. The explicit decoupling between user's goals and capabilities provides the system the freedom to generate the orchestration plan at run-time, according to the contextual state. An industrial case study, conducted in for a scientific project, has provided the conditions for evaluating the running example of a B2B business process for a fashion enterprise.

1 Introduction

Cloud Computing, Internet of Things and Internet of Everything will likely be the base for a new technological revolution in science, business, trade, production and social processes. Cloud Computing, in fact, represent a fundamental change in the way information technology services are invented, developed, deployed, scaled, and updated. Moreover, the payment formula pay-by-use make Cloud Computing services attractive for all enterprises that do not have Information Technology activities as core business.

L. Sabatucci, S. Lopes and M. Cossentino
ICAR-CNR
E-mail: `{sabatucci,s.lopes,cossentino}@pa.icar.cnr.it`

The current trend shows that, in the next future, cloud applications will be ubiquitous, available over the Internet [22] and could provide the most application as a single or composition of multiple services. Then, enterprises able to satisfy customer expectations in a quickly changing environment can be the winner in business competition [35]. These companies require indeed ways to make their processes more flexible and, at the same time, to open their business processes to the direct access of users. In such a perspective, technology can play a crucial role allowing organisations to adopt agile methodologies. In particular, Cloud computing focuses on maximising the effectiveness of shared resources and information (provided to users on-demand) reducing the overall cost by using less power, air conditioning, rack space, and so forth. However, Cloud applications are still currently developed as closed solutions linked to proprietary architectures in which the provider furnish all elements of the service [22]. This feature represents an obstacle for third-party developers. Thus, it results difficult to mix and match cloud services from different levels and to configure them dynamically to address application needs [22].

Cloud Application Mashup has the aim to enable easier customization and composition of SaaS applications from several providers. It will provide a cohesive solution that offers improved functionality to clients. The mashup task involves data and process integration complex protocols and at the same time preserves global application consistency.

So far, web developers have used application mashup as a tool for integrating content from more than one source in a new single graphical interface. For example, HousingMaps (http://www.housingmaps.com), combines rental listings from a popular advertisements website with Google Maps for providing a visual representation of local apartments for rent. Other examples are The New York Times (http://www.nytimes.com), CNN.com (www.cnn.com) and BBC (http://www.bbc. com). All these magazines include buttons for social sharing. Web services, running on Cloud SaaS, as Google Applications, Dropbox, Microsoft Office365 are today used by millions of users. The mashup technology is still evolving for mixing processes together with data and user interfaces to produce more sophisticated applications in simpler and quicker ways.

We believe that the consuming users can create their Mashups using the adequate support. This paper presents an approach for generating Cloud Application Mashups automatically. So that, users can profit from software licensing and billing models based on the pay-for-use formula. Cloud services should be available from the public cloud marketplace in which providers store their offerings. Clients can discover and buy the needed services to use for their mashup. They will select, mix and integrate the necessary services from a variety of cloud providers. Therefore, a mashup self-composition engine acting as a run-time mediator between users goals and atomic cloud services would be a very useful tool for realising the desired application.

We already developed GoalSPEC [27], a goal-oriented [34] language for describing the expected behaviour of a complex distributed system. We also presented [24] an algorithm for automatically selecting and aggregating services to address users goals. However, this algorithm relies on a centralised logic and does not scale up well with the goal and service number.

The main novelty of the current paper is the formulation of a three-layer architecture for implementing Cloud Capabilities and supporting cloud application mashup. We define cloud capability a smart and autonomous container of traditional web services. It is an application running at Saas level inspired by the principle of autonomic computing. It senses the environment, makes proactive decisions, and interact with other cloud capabilities for producing a coordinate behaviour. They must be provided together with relevant aspects of their integration and usage through ad-hoc description languages thus enabling the automatic composition of their services. To this aim, we developed a capability description language that exploits predicate logic for specifying when and how to use a service. Decoupling the specification of what the system has to do from how this what will be done allows the self-configuration engine to compose the expected behaviour on the occurrence. We report the case of a cloud mashup built for a fashion enterprise as running example where we described the desired process through a set of goals. Thus, the submission of the input goals becomes a stimulus for the system to configure ad-hoc solutions. Obviously, we also populated the marketplace with a redundant set of services for addressing the requested behaviour. Self-configuration protocols achieve the expected B2B mashups using the relevant cloud capabilities. Self-configured Mashup is a facilitator for fast and flexible B2B collaboration (short development cycles, cheap development) whereas existing B2B collaboration solutions focus on long-term business relationships [28]. Therefore, we think that our approach could have a profound impact on IT by improving the return-on-assets of existing systems. In fact, quick development cycles make B2B solutions attractive for small and medium enterprises.

The organisation of the paper is the following: Section 2 presents the main concepts of the approach: state of the world, world transition system, capability, goal and configuration. Section 3 presents a real scenario occurred during a research project and introduced the elements for enabling the self-configuration. Section 4 illustrates details of the self-configuration module: the three-layers architecture and the cloud capability. Section 6 focuses on the self-configuration distributed process, whereas Section 6 explains the dynamic workflow generation related to a mashup. Related works are discussed in Section 7 whereas we report some conclusions and acknowledgment in Section 8 and Section 9.

2 Background

In this section, we describe some of the main concepts presented in the paper.

A *State of the World* is an abstraction made to let the system reasoning at the knowledge level [21]. It arises from the consideration that a software system has (partial) knowledge about the environment in which it runs. The classic way of expressing this property is (Bel a φ) [32] that specifies that a software agent a believes φ is true, where φ is a fact that describes a particular state of affair. This concept is similar to the idea of the fluent found in the situation calculus. The definition of relevant and coherent states of the world requires a preliminary analysis of the domain that ontology perfectly captures by means of the description of classes, properties, and individuals of interest The proposed approach is independent on the instrument for modelling the domain (e.g. OWL-S [19], POD [8]).

Definition 1 (State of the World) The state of the world at a given time t is a set $W_t \subset S$ where S is the set of all the non-negated facts $s_1, s_2 \dots s_n$ that characterises a given domain.

W_t has the following characteristics:

$$W_t = \{s_i \in S | (\text{Bel } a \; s_i)\} \quad (1)$$

where a is the subjective point of view (i.e. the execution engine) that believes all facts in W_t are true at time t.

Whereas a state of the world depicts assertions that are true at a precise time instant (ex: P is true), the World Transition System (WTS) is the structure used for describing how the truth values of assertions may change over time in an indeterminist environment [9].

Definition 2 (World Transition System) A World Transition System (WTS) is a tuple $\langle \Sigma, W_0, C, E, \mathcal{L} \rangle$ where:

- $\Sigma \subseteq 2^S$ is the finite set of states of the world, where S is the set of all the facts that are designated for the domain of interest;
- $W_0 \in \Sigma$ is the initial state of the world;
- E is the transition relation defined in $\Sigma \times \Sigma$; each transition is labelled with the capability that originates the evolution;
- x is the relation defined in $E \times E$ that designates a not deterministic link between two transitions.

The semantic of this structure is to model the execution of a system of concurrent processes: the computation starts from the initial state of the world and may follow anyone of some different paths in the graph (sequences of states of the world).

A *Capability* describes a self-contained process, which can modify the current state of the world. Each capability represents a semantic wrapper for describing existing cloud applications and services.

Definition 3 (Capability) A capability $\langle E, pre, post \rangle$ is an atomic and self-contained action the system may intentionally use to address a given evolution of the state of the world. E is a set of possible evolution scenarios, each denoted as $evo : W \rightarrow W$ is an endogenous change of the state of the world that takes a state of the world W_t and produces a new, different state of the world W_{t+1}. The capability may be executed only when a given pre-condition is true ($pre(W_t) = true$). Moreover, the post-condition is a run-time helper to check if the capability has achieved the desired condition($post(W_{t+1}) = true$).

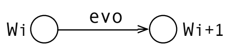


Fig. 1 Representation of an evolution scenario in a world transition system.

An evolution scenario describes the effects of executing the capability, i.e. the global state changes from an initial state W_i towards the resulting W_{i+1}.

The presence of more evolution scenarios in a capability implies that the result is not deterministic at run-time. The real output depends on the actual execution context, the input value or other external variables (for instance a human decision).

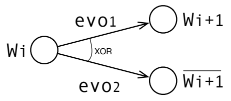


Fig. 2 Representation of a set of not deterministic evolution scenarios in a world transition system.

It is worth noting the use of capabilities has the advantage of facilitating service composition in order to address a complex result.

The concept of *Goal* is often used in the context of business processes for representing strategic interests of enterprises that motivate the execution of a particular workflow [34]. It is "the desired *change* in the state of the world an actor wants to achieve".

Definition 4 (Goal) A goal is a pair: $\langle tc, fs \rangle$ where tc and fs are conditions to be evaluated over a state of the world. Respectively the tc describes when the goal should be actively pursued and the fs describes the desired state of the world. Moreover, given a W_t we say that

$$\text{the goal is } \textit{addressed} \text{ iff } tc(W_t) \wedge fs(W_{t+k}) \text{ where } k > 0 \quad (2)$$

i.e. a goal is addressed if and only if, given the trigger condition is true, then the final state must eventually hold true some time later on the temporal line.

The user should specify his mashup application through a goal set.

The ***Operationalization*** of a goal is an orchestration plan that is used to address the given result. We adopted a notation for representing plans inspired by the workflow pattern initiative work [31] that aims at providing a conceptual basis for business process technology. In particular, the definition of a plan is described with a set of tasks, organised through three main control flow patterns: sequence, exclusive-choice, and structured-loop.

In a *sequence*, the completion of a task enables the next one in the same process. The Sequence pattern serves as the fundamental building block for processes. It is used to construct a series of following tasks which execute in turn one after the other (see Figure 3).

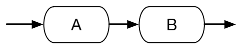


Fig. 3 The Sequence workflow pattern.

In an *exclusive-choice* (also known as XOR-split), a branch splits into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one of the outgoing branches based on a design-time condition (see Figure 4).

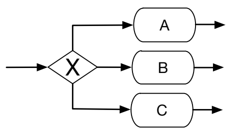


Fig. 4 The Exclusive-Choice workflow pattern.

A *structured-loop* describes the ability to execute a task repeatedly. The loop is characterized by a condition that is either evaluated at the beginning or the end of the loop (see Figure 5). The looping structure has a single entry and exit point.

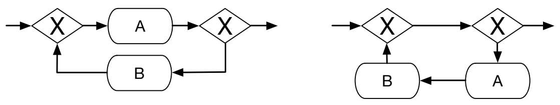


Fig. 5 Two alternative versions of the Structured-Loop workflow pattern. On the left, the loop is characterized by a post-test condition. On the right, the loop is characterized by a pre-test condition.

We define a ***Configuration*** as a set of tuples of type $\langle g, h \rangle$, where g is a goal to be addressed and h is the operationalization for that goal. In a valid configuration 1) each input goal is fully achieved by a plan for the fulfilment; 2) there is not redundancy: each goal is associated exactly to one operationalization.

Generally, given a set of goals, a set of capability, and an initial state, it is possible to derive zero or more configurations. The self-configuration phase consists in 1) incrementally building a state transition system [[23] that models the many possible worlds that are reachable by using the available capabilities, and, subsequently, 2) extracting alternative configurations. The next section illustrates an example of cloud mashup.

3 An Example of Cloud Application Mashup

This section provides an overview of the self-configuration process with the help of a running example extracted from the activities conducted in the OCCP research project.

3.1 The Open Cloud Computing Platform Project

The objective of the OCCP project[1] was to deliver a cloud architecture for the mashup of distributed applications. The mission was to provide a market-oriented infrastructure for creating cloud applications by merging existing services available at the SaaS layer. The project aimed at proposing a new dynamic approach to business models in which independent, proactive and self-adaptive processes solve composition/orchestration problem.

Figure 6 depicts the primary objectives of the project. It encompasses the business process modelling as a twofold activity. The first side concerns with the analysis of business processes. The counterpart deals with the analysis of heterogeneous data coming from various parties. This first activity leads to a couple of sub-activities: process mashup design and data mashup design. The former provides a high-level description of the business process to be generated regarding goals, i.e. what systems users expected. The former produces a high-level description of services data, using their symbolic expression. Both the activities collaborate in generating a common ontology for denoting the business semantics. Whereas all these activities are manual tasks, the goal-oriented self-configuration is automatic. It takes all the previous outputs, plus a service repository and a service description repository, and generates/executes the resulting cloud application mashup.

We adopted this approach in an industrial case study delivered in a real cloud environment. Each industrial partner provided a set of services to be integrated: 1) eCommerce services, 2) Invoice Manage- ment services,

[1] Funding program PO FESR 2007-2013 Sicily, started in 201- ended in 201-. The project web page is available at `http://aose.pa.icar.cnr.it/OCCP/`

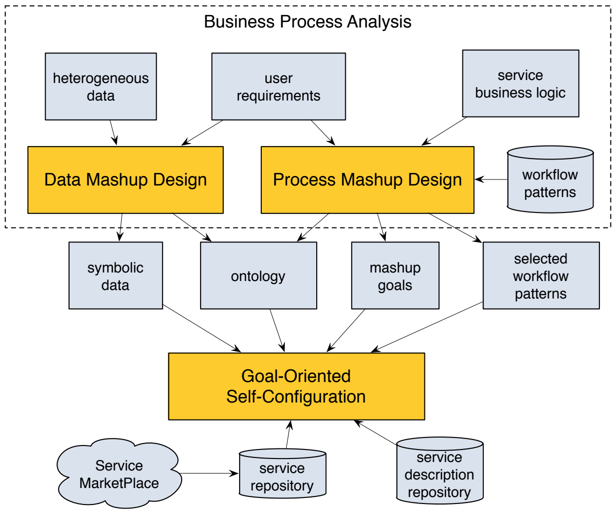
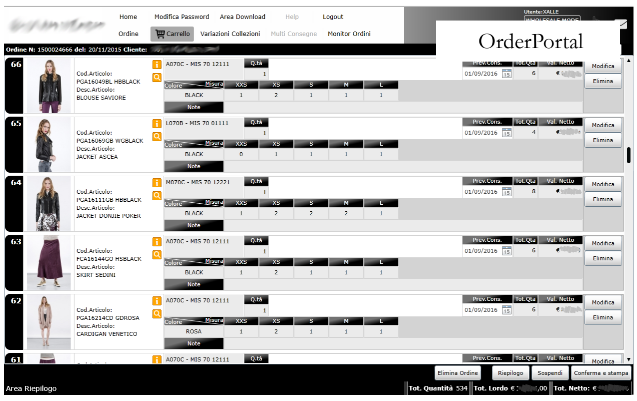


Fig. 6 The objectives of the OCCP project.



[redacted] Home Modifica Password Area Download Help Logout Utente:XALLE
Ordine Carrello Variazioni Collezioni Multi Consegne Monitor Ordini
OrderPortal

Ordine N: 1500024666 del: 20/11/2015 Cliente: [redacted]

	Cod.Articolo	Desc.Articolo		Q.tà	Colore	XXS	XS	S	M	L	Prev.Cons.	Tot.Qta	Val. Netto	
66	PGA16049BL HBBLACK	BLOUSE SAVIORE	A070C - MIS 70 12111	1	BLACK	1	2	1	1	1	01/09/2016	6	€ [redacted]	Modifica Elimina
65	PGA16069GB WGBLACK	JACKET ASCEA	L070B - MIS 70 01111	1	BLACK	0	1	1	1	1	01/09/2016	4	€ [redacted]	Modifica Elimina
64	PGA16111GB HBBLACK	JACKET DONJIE POKER	M070C - MIS 70 12221	1	BLACK	1	2	2	2	1	01/09/2016	8	€ [redacted]	Modifica Elimina
63	FCA16144GO HSBLACK	SKIRT SEDINI	A070C - MIS 70 12111	1	BLACK	1	2	1	1	1	01/09/2016	6	€ [redacted]	Modifica Elimina
62	PGA16214CD GDROSA	CARDIGAN VENETICO	A070C - MIS 70 12111	1	ROSA	1	2	1	1	1	01/09/2016	6	€ [redacted]	Modifica Elimina
61			A070C - MIS 70 12111											Modifica

Elimina Ordine Riepilogo Sospendi Conferma e stampa
Area Riepilogo Tot. Quantità 534 Tot. Lordo € [redacted],00 Tot. Netto: € [redacted]

Fig. 7 Screenshot of the website hosted and managed by the OrderPortal Capability.

3) DataBase and File Storage services, and, 4) Voicemail and Notification services. The follow subsections illustrate the whole case study.

3.2 A B2B Cloud Application for Fashion Firms

A world known fashion enterprise, here named FashionFirm for privacy reason, uses a legacy system (IBM AS/400) for managing its information system. To the aim of enlarging its commercial network, FashionFirm designated a small software house, denoted as SWHouse, for handling its B2B processes. SWHouse developed a system on a set of services running on a cloud stack. That is a set of scalable backend services able to interact with the legacy system from one side and with a SaaS eCommerce platform (OrderlPortal) from the other side. SWHouse was also demanded to enrich the FashionFirm business process by adding new services for customer management. These new services were conceived as a mashup of cloud application with the aim to improve the costs-benefit ratio. In fact, this allowed SWHouse to fast prototype the solution reusing already existing cloud application provided by third parts (Cloud Calendar, File Storage, Voicemail ...). The resulting mashup application that we use as running example in this paper is designed for supporting customers during the order management process. In particular, RetailStore is a retailer of FashionFirm products. When RetailSore requests for a product stock through the OrderPortal, the system merges the legacy services with external applications provided by cloud computing providers. Moreover, the resulting application employed a Cloud Storage system for storing and delivering receipts to RetailSore, Voicemail for conveying a recorded audio message to a recipient, finally, a Cloud Calendar service for annotating the delivery status.

3.3 Elements for Self-Configuration

The domain expert had to choose the entities and the related properties necessary to model the above scenario. This conceptual model of a domain is rendered as an ontology, and it is composed of ***entities***, entity's ***properties*** and ***is-a***, ***has-a*** relationships. The entities and related properties used in the running example are reported in the following list:

```
entities:
   role, document, url, failure_order
   user [is-a role],
   storehouse_manager [is-a role],
   order [is-a document],
   invoice [is-a document],
   user_data [is-a document],
   registration_form [is-a-document],
   delivery_order [is-a order],
   rxfile [is-a URL]

unary user's properties:
    registered,
    unregistered,
    has_cloud_space

unary document's properties:
    available, uploaded_on_cloud

unary order's properties:
    accepted, refused, processed

unary user_data's properties:
    complete, uncomplete

unary registration_form's properties:
    complete, uncomplete

binary properties:
    notified(document,user),
    mailed_perm_link(document,user) [is-a notified]
```

For instance, the class user denotes FashionFirms customers, whereas the class order denotes a complex description of items and quantity that a customer is going to buy from FashionFirm. Properties and is-a predicates are translated into domain rules, i.e. 'user [is-a role]' is translated into the rule 'role(X) :- user(X)'.

The developer uses the ontological description of the scenario for defining the desired mashup using of a set of ***goals*** to pursue. An automatic tool for deriving the Goal decomposition from an ontological model is reported in [8], whereas a language for describing GOAL is reported in [27]. BPMN2GoalSPEC is an automatic tool for generating the goals from BPMN[2]. We report the corresponding GoalSPEC definition for the running example in the following list

```
GOAL to_wait_order:
    WHEN MESSAGE X RECEIVED FROM THE Client ROLE
         AND order(X) AND user(Client)
    THE SYSTEM SHALL ADDRESS
    available(X) AND order(X)

GOAL to_notify_invoice:
    WHEN accepted(X) AND order(X) AND registered(
         Client) AND user(Client)
    THE SYSTEM SHALL ADDRESS
    MESSAGE Z SENT TO THE Client ROLE AND user(
         Client) AND invoice(Z)

GOAL to_deliver_order:
    WHEN MESSAGE X SENT TO THE Client ROLE AND
         invoice(X) AND user(Client)
    THE SYSTEM SHALL ADDRESS
    MESSAGE Z SENT TO THE SM ROLE AND
         delivery_order(Z) AND storehouse_manager(
         SM)

GOAL to_notify_failure:
    WHEN refused(X) AND order(X) AND registered(Y)
         AND user(Y)
    THE SYSTEM SHALL ADDRESS
    MESSAGE failure_order SENT TO THE Client ROLE
         AND user(Client)
```

In the above listing, all the words in upper case are keywords of GOALSpec language whereas the words in lower case are entities or properties anchored to the ontological description. The GOALSpec description specifies what should be done for ensuring the correct execution of the order management process.

It is obvious that there must be a cloud application or a combination of cloud applications that effectively satisfies each single goal. Moreover, the self-configuration requires additional information about available services for automatically selecting and binding services to users goals.This information is provided by the means of the ***capability*** language.

First, the order must be processed via the eCommerce website (`order_portal`). The *OrderPortalMonitor* Capability is responsible for hosting the website and to wait for new orders from users.

```
CAPABILITY OrderPortal:
 Pre-Condition: --
 Post-Condition: available(X) & order(X)
 Evolution: [add(available(an_order))]
```

[2] BPMN2GoalSPEC is available as a web-service at `http://aose.pa.icar.cnr.it:8080/BPMN2GOALSPEC/`

A screenshot of the eCommerce website managed by this capability is shown in Figure 7.

The availability of products in the FashionFirm storehouse must be checked by the *CheckStoreHouse* capability.

```
CAPABILITY CheckStorehouse:
 Pre-Condition: available(X) & order(X) &
     registered(X) & user(X)
 Post-Condition: ( accepted(X) | refused(X) ) &
     order(X)
 EvolutionSet:
  AcceptableOrder: [add(accepted(an_order)), remove
      (available(an_order))],
  UnacceptableOrder: [add(refused(an_order)),
      remove(available(an_order))]
```

CheckStoreHouse is an example of non-deterministic capability: it produces two possible final states where the order has been either accepted or refused. If the requested products are available, the corresponding invoice should be delivered through the *UploadOnCloudStorage*. Otherwise, the invoice is stored locally, and a link will be communicated to the user via mail. Figures 8 and 9 show respectively the *UploadOnCloudStorage* working with the commercial Dropbox API[3] and the *ShareFileLink* that uses Google Drive services[4] for sending the notification.

```
CAPABILITY UploadOnCloudStorage:
 Pre-Condition: available(X) & invoice(X) &
     has_cloud_space(Y) & user(Y) & !
     uploaded_on_cloud(X)
 Post-Condition: upload_on_cloud(X) & invoice(X)
 Evolution: [add(uploaded_on_cloud(the_invoice))]

CAPABILITY ShareFileLink:
 Pre-Condition: uploaded_on_cloud(X) & invoice(X) &
      NOT has_cloud_space(Y) & user(Y) & !
     mailed_perm_link(X, Y)
 Post-Condition: mailed_perm_link(X,Y) & invoice(X)
      & user(Y)
 Evolution: [add(mailed_perm_link(the_invoice,
     a_user))]
```

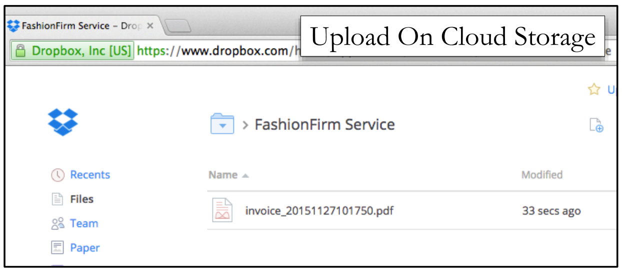
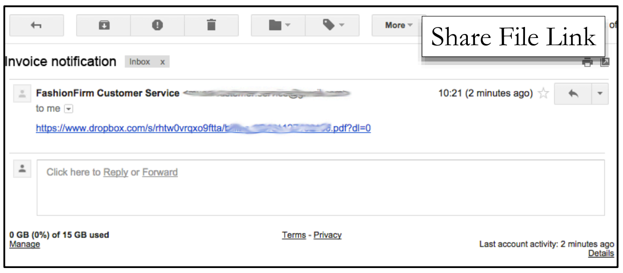

FashionFirm Service – Drop
Dropbox, Inc [US] https://www.dropbox.com/
Upload On Cloud Storage
FashionFirm Service
Recents
Files
Team
Paper
Name
Modified
invoice_20151127101750.pdf
33 secs ago

Fig. 8 Screenshot of the front-end of the *UploadOnCloudStorage* Capability implemented through the Dropbox services.

We describe in more detail the link between capabilities and web services. During the self-configuration

[3] `https://www.dropbox.com/developers-v1/core/docs`
[4] `https://developers.google.com/drive/v2/reference/`



Invoice notification Inbox x

FashionFirm Customer Service [redacted] 10:21 (2 minutes ago)
to me
https://www.dropbox.com/s/rhtw0vrqxo9ftta/[redacted].pdf?dl=0

Click here to Reply or Forward

0 GB (0%) of 15 GB used
Manage
Terms - Privacy
Last account activity: 2 minutes ago
Details

Fig. 9 Screenshot of the front-end of the *ShareFileLink* Capability realized through the Google Apps service.

phase, the system automatically checks all the available capabilities in the repository for their compatibility to the requested goal. The way the automatic self-configuration happens is reported in Section 5. The chosen capabilities are linked to the corresponding web services during the orchestration phase (Section 6).

The self-configuration phase builds the world transition system by exploiting available capabilities.

A relevant part of the transition system for the B2B scenario is reported in Figure 10. This graph's nodes represent possible (stable) states of the system. Each node of this graph represents possible (stable) states of the system. The transition from a state to another one is either due to an external even (denoted with <=) or due to the execution of a web service (internal event, denoted with !). Non-deterministic transitions are highlighted through a xor operator that links two or more transitions. Initially, the system is in a state in which it waits for user orders from the `order_portal` service. When an order incomes, if the user is already registered, then the system retrieves user's data from a database. Otherwise, has two possibilities: 1) initializing a registration procedure (`send_reg_form` and `register_user` services) or 2) manually put the user's data into the database (this requires an employee that manually processes the request form.

Subsequently, the order is checked over the storehouse database (`check_storehouse` service) thus to be processed until all the ordered products are ready to be delivered (`notify_storehouse_manager` service).

An interesting part of Figure 10 is that in which the invoice is ready to be sent (after the `generate_invoice` service) and the system must send it to the user. The system can choose: 1) to directly upload the file on the users cloud storage (`upload_on_cloud` service) 2) to locally store the file and to send a mail containing the file's URI (obtained by combing the `upload_on_cloud` and the `share_link_file` services).

We highlight this situation because, in points like this, the system must be able to generate different behaviours for addressing the same goals. Configurations capture this kind of variability that is fundamental to customise the final system.

Figure 11 shows three different configurations (among the many possible), resulting from the self-configuration phase of the running example.

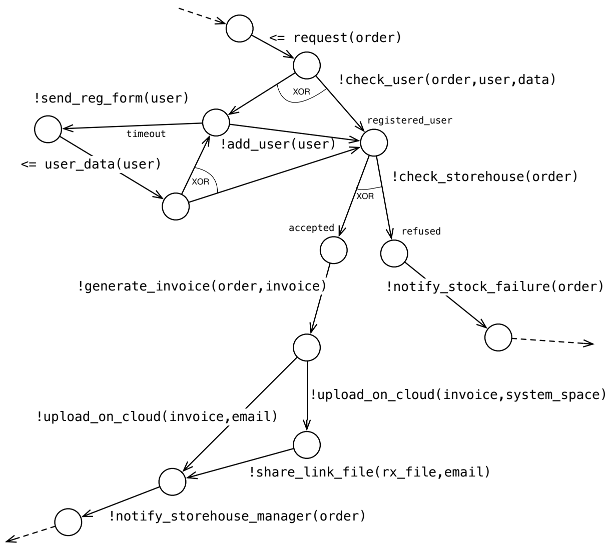

<= request(order)
!check_user(order,user,data)
XOR
!send_reg_form(user)
registered_user
timeout
!add_user(user)
<= user_data(user)
XOR
!check_storehouse(order)
XOR
accepted
refused
!generate_invoice(order,invoice)
!notify_stock_failure(order)
!upload_on_cloud(invoice,system_space)
!upload_on_cloud(invoice,email)
!share_link_file(rx_file,email)
!notify_storehouse_manager(order)

Fig. 10 World transition system for the B2B scenario. Transitions represent evolutions due to cloud capabilities. Indeterministic transitions are represented with an 'xor bridge' i.e. an arc connecting two or more evolutions. Prefixes ! and <= indicate respectively an internal event (i.e. the execution of a service) and an external event (i.e an incoming message).

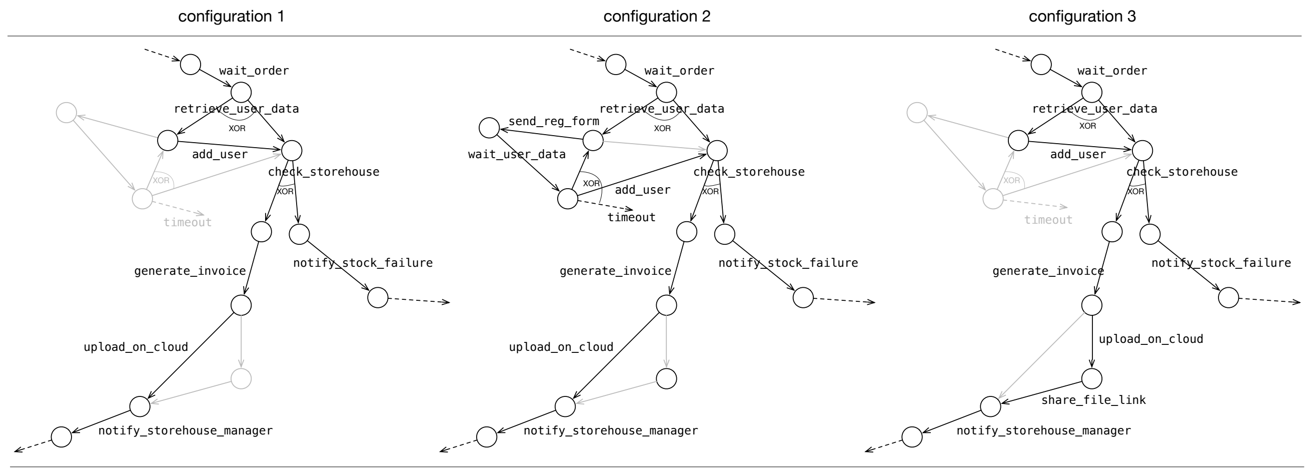
4 The Proposed Architecture for Cloud Mashup

This section illustrates an approach to domain-independent self-configuration of cloud applications. Self-configuration is intended as the ability to automatically aggregate and configure a set of services thus to ensure the correct execution for achieving the defined users goals [30].

4.1 A Three-Layered Architecture for Self-Configuration

The proposed approach is structured in three interoperating functional layers: the goal layer, the capability layer, and the service layer.

The uppermost layer of this architecture is the ***Goal Layer*** in which the user may specify the expected behaviour of the system regarding high-level goals. Goals are not hard-coded in a static goal-model defined at design time. The goal injection phase allows the introduction of user-goals defined at run-time. Goals are in-



```
to_wait_order <-- sequence {
                    !wait_order,
                    !retrieve_user_data
                }

to_notify_invoice <-- sequence {
                    !check_storehouse,
                    exclusive-choice {
                        sequence {
                            !generate_invoice,
                            !upload_on_cloud
                        },
                        !notify_stock_failure
                    }
                }

to_deliver_order <-- !notify_storehouse_manager

to_notify_failure <-- !notify_stock_failure
```

```
to_wait_order <-- sequence {
                    !wait_order,
                    !retrieve_user_data,
                    structured-loop {
                        !send_reg_form,
                        !wait_user_data
                    }
                    !add_user
                }

to_notify_invoice <-- sequence {
                    !check_storehouse,
                    exclusive-choice {
                        sequence {
                            !generate_invoice,
                            !upload_on_cloud
                        },
                        !notify_stock_failure
                    }
                }

to_deliver_order <-- !notify_storehouse_manager

to_notify_failure <-- !notify_stock_failure
```

```
to_wait_order <-- sequence {
                    !wait_order,
                    !retrieve_user_data
                }

to_notify_invoice <-- sequence {
                    !check_storehouse,
                    exclusive-choice {
                        sequence {
                            !generate_invoice,
                            !upload_on_cloud,
                            !share_file_link
                        },
                        !notify_stock_failure
                    }
                }
```

Fig. 11 The first three configurations that have been identified for for the B2B scenario. In the top side it is reported the subgraph of the WTS that originates the configuration (the remaing part of the graph is shaded in gray). Below, the configuration is reported by highlighting the three workflow patterns (sequence, exclusive-choice and structured loop).

terpreted and analysed and therefore trigger the need for the system to generate a new configuration.

The second layer is the ***Capability Layer***, based on solving at run-time the problem of Proactive Means-End Reasoning [24].

It aims at selecting the capabilities (and configuring them) as a response to requests defined at the top layer. This task corresponds to a strategic deliberation phase in which decisions are made according to the (often incomplete) system knowledge about the environment. The output is the selection of a set of capabilities that will form a correct and effective business process. This is obtained by instantiating system capabilities into business tasks and by associating capability parameters with data objects. In this phase, the procedure also specifies dependencies among tasks and how data items are connected to task input/output ports.

The third layer is the ***Service Layer***, it manages and interconnects autonomous blocks of computation thus generating a seamless integration for addressing the desired result specified at the first layer. Section 6 describes the run-time orchestrator that executes the business process generated at the second layer by interacting with the corresponding cloud applications and web-services.

4.2 Implementing the Cloud Capabilities

The aforementioned high-level architecture is deployed through a distributed system of software entities namely ***Cloud Capabilities***.

Whereas traditionally services and cloud applications are passive entities that act when receiving the control [19, 20], cloud capabilities are lively cloud applications (running at SaaS) inspired to the principles of autonomic computing. Each cloud capability keeps its own control, and it is able of sensing the surrounding environment, making proactive decisions, and interacting with other capabilities for organising a common behaviour [24, 26].

From the developer's point of view, a cloud capability is a facilitator for quickly implementing cloud applications with high-level features such as autonomy, proactiveness, self-organization and logic reasoning. In a mashup, a capability acts as a stateful wrapper for a specific web service or a cloud application.

We implement a cloud capability through two components as shown in Figure 12: a general-purpose, reusable core and a service customised part. The core provides a generic set of APIs to support the three-layered architecture shown in the previous section:

- Goal Management: some facilitators to handle the interpretation of single goals in GoalSPEC and to manage the whole goal-set;
- a Self-Configuration module that exploits a logic-based reasoner (to handle the matching between goals and capabilities) and some self-organization protocols (to collaboratively generate configurations);
- an Adaptive Orchestrator responsible of translating a configuration into an operative plan and to enact the corresponding workflow in a dynamic environment.

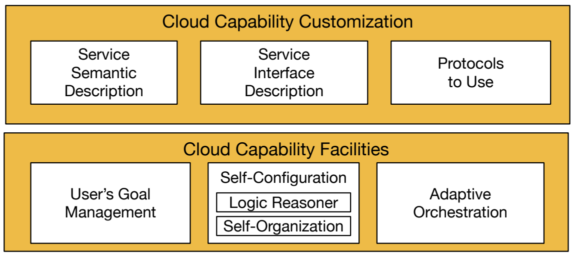

Cloud Capability Customization
Service Semantic Description
Service Interface Description
Protocols to Use
Cloud Capability Facilities
User's Goal Management
Self-Configuration
Logic Reasoner
Self-Organization
Adaptive Orchestration

Fig. 12 Internal Architecture of a Cloud Capability.

The customizable part of a capability allows to set up the generic 'core' for working with a specific service (either web-service or cloud application). The *Service Interface Description* allows specifying how to manage the service wrapping (service input/output ports), including how to maintain a state for the correct functioning of stateful services. The *Protocol to Use* (HTTP/ HTTPS/ SOAP ...) may be selected among pre-defined ones, even if new ones may be programmed from scratch to occurrence. Finally, the *Service Semantic Description* allows to specify information related to the self-configuration phase such as pre/post conditions and evolution (as described in Section 2).

For instance, *UploadOnCloudStorage* is a wrapper for a generic REST file storage service on the cloud (`upload_on_cloud(file,user_account)`) that requires defined parameters and produces a remote clone of a local file by returning a unique id that identifies that remote object. The following first-order predicate represent the *cloud capability customization* part for the capability as mentioned earlier.

```
capability(upload_on_cloud_storage,

    % semantic description
    precondition(available(document)),
    postcondition(uploaded_on_cloud(document)),
    evolution(add(uploaded_on_cloud(document))),

    % service interface
    method(https,put),
    address(https://content.dropboxapi.com/1/
        files_put/auto/),
    input([
          param(local_file,file),
          param(remote_file_path,rxfile)]),
    output([param(metadata,json_file_descriptor)]),

    % protocols
    service_protocol(rest),
    require_auth(oauth20)
).
```

The first compartment (*semantic description*) contains exactly information already discussed in Section 3.3: pre/post conditions and evolution. The second compartment (*service interface*) specifies which method to use for the service invocation, the address for reaching the service and any input/output ports to provide for a correct invocation. Finally, the last compartment (*protocols*) indicates that the service call must follow the REST protocol and the consequent request should be signed (it requires a preliminary OAuth authentication).

5 Self-Configuring a Mashup

This section focuses on the middle layer of the presented architecture, and in particular on the strategy adopted for automatically establishing run-time links between user's goals and the available capabilities. Please, refer to [27] and [26] for more details about GoalSPEC (goal layer) and self-adaptation (service layer), respectively.

Here the problem is, for each goal g_i, to discover which combination of capabilities (according to the workflow patterns) may be employed for satisfying Equation 2.

In other words, invoking a single service produces changes in the state of the world that are specified by the corresponding capability's property *evolution*. This property describes the expected changes with *add* and *delete* operators that respectively add new statements to the state of the world, or delete existing statements, for producing the resulting state.

Consequently, executing a workflow composed of capabilities produces a multi-step evolution of the state of the world, i.e., $e = \{W_1, W_2, \dots, W_n\}$. The evolution e satisfies a goal when the goals' trigger condition is satisfied in $W_{k1} \in e$, and the final state eventually holds later in some subsequent state: $W_{k2} \in e : k2 \geqslant k1$.

5.1 The Decentralized Strategy for Self-Configuration

The possible evolution paths of a system are modelled as a state transition system (WTS, hereafter) where nodes are states of the world and transitions are due to the execution of capabilities (see Figure 10). Our system builds the WTS with the contribution of the available cloud capabilities.

The WTS is implemented as a blackboard cloud service, i.e. accessible by all the capabilities, so that every cloud capability may add new nodes and transitions in a collaborative fashion. However, to avoid the concurrent modification of the same WTS, cloud capabilities enforce a blind auction protocol [16] for deciding the priority of write access, as described herein.

The blackboard service allows users to register a new goal set. When this happens, it creates a new shared WTS that only contains the initial state of the world node. Subsequently, it starts a cycle of auctions, playing the role of auctioneer, and it periodically sends a call-for-bids to any potentially interested capability.

At the same time, each cloud capability starts an *expand-and-evaluate* cycle, working on the portion of WTS already available. They concurrently produce new states and transitions and store them in their privately memory space. These new states are evaluated according to a global scoring function. This evaluation is used for predicting how much the new state is promising in respect to the whole goal-set.

Periodically, when new call-for-bid incomes, each cloud capability selects the best state, it has generated during past expand-and-evaluate iterations. The state score is used for setting a bid for participating in the auction. The auction winner gains the permission to update the WTS.

This strategy rewards those capabilities that promise to improve the WTS by increasing the global goal satisfaction.

5.2 The Expand-and-Evaluate Cycle

When a user specifies a goal set to address, cloud capabilities enters in an expand-and-evaluate cycle (Figure 13). During this phase, the capability reads the global WTS and generates local expansions for the graph. Clearly, expansions are not synchronised among the several capabilities. The cycle is described in the following.

1. Each capability selects those nodes of the WTS that satisfy its pre-conditions;
2. The capability picks the most promising node, among the selected ones (Figure 13.a);

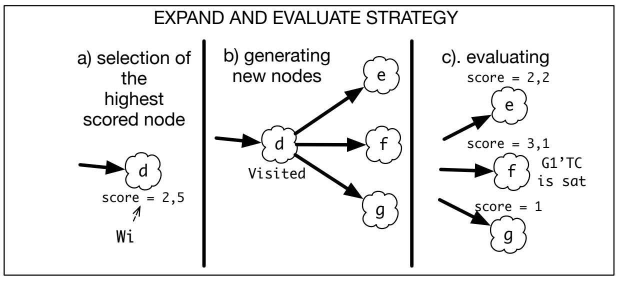


Fig. 13 Steps of the Expand and Evaluate Strategy.

3. The capability simulates the effects of its wrapped service by generating new states through the evolution property (Figure 13.b).
4. The capability generates a score for each new node and stores them in a private data structure. The more the state of the world is close to addressing some goals, the higher is the score assigned to it (Figure 13.c). A node is also marked as TC_holds_i or FS_holds_i if the node satisfies respectively the triggering condition TC_i or the final state FS_i of the goal g_i. More details about the score computation may be found in the following subsections.

5.3 The Global Scoring Function

The aim of the scoring function is to predict how much a state of the world is *close* to the final state where a goal is satisfied. A state of the world is described by a finite set of statements. We defined a function that evaluates the potential effect of each of these statements for addressing a goal. We defined a function to the aim of producing a quantitative measurement. In fact, that function rewards statements that provide a positive impact to a goal, and it penalises statements that are less effective. The function is defined as follows:

$$score(W) = \sum_{g_i} \frac{1 + num\_rel\_stats(W, g_i)}{num\_stats(W)} \qquad (3)$$

where, given a state W, $num\_rel\_stats(W, g_i)$ is defined as the number of statements contained in $W \cap (TC_g \cup FS_{g_i})$, whereas $num\_stats(W)$ is the cardinality of W, i.e. the number of statements contained in W. For instance, if $W = \{s_1, s_2, s_3, s_4, s_5\}$ and $g = \langle s_2 \wedge s_8, s_4 \vee s_5 \rangle$ then $num\_stats = 5$ and $num\_rel\_stats = 3$ because $\{s_2, s_4, s_5\}$ are relevant for g.

Figure 14 illustrates Function 3 plotted as a stacked line chart for highlighting the score trends. Setting to constant the *num_stats* in the formula, the score is higher the more the statements are relevant for the

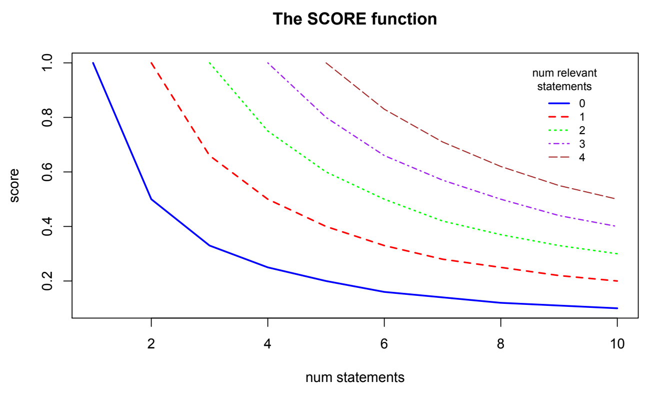


Fig. 14 Line chart of the score function highlights trends of the value when making either $num\_stats(W)$ or $num\_rel\_stats(W, g)$ constant.

goal satisfaction. Conversely, making the num_rel_stats constant, the score increases when the total number of statements in W decreases.

This may be interpreted as follows: a state is interesting if it promises to converge quickly to the desired final state. Clearly, there is not warranty the prediction based on this heuristic is either optimal or perfect, but it empirically proved to be sufficient for the specific purpose of speeding up the exploration of the graph.

5.4 The Auction Cycle

The auctioneer launches a new blind auction every constant time intervals. The four steps of the auction phase are shown in Figure 15.

The auction starts and a call-for-bid is sent to all the capabilities. They have a fixed deadline to reply with their bid (see Figure 15.a).

Each participant selects the node –from its private expand list– with the highest utility, calculated as it follows: the node's score plus the number of TC_holds and FS_holds for that state. The highest utility is the bid to be sent back to the auctioneer (see Figure 15.b).

There is not counter-offer, the auction closes when all capabilities replied or at a predefined deadline. The highest bid wins the auction (see Figure 15.c).

Consequently, the winner pulls the selected node from its private data structure to the global WTS, also reporting eventual transitions with pre-existent nodes. For instance, Figure 15.c shows that the new node f is connected with its predecessor d from which it originated.

The procedure cycles again with a new auction until a MAX number of configurations is discovered or if the auctioneer receives only empty offers for a fixed number of times (expanding the WTS is no more possible).

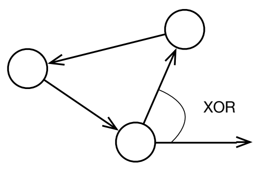
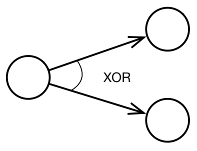
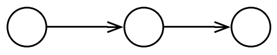
6 From the Transition System to the Mashup

This section describes how a configuration, extracted from the WTS, is transformed in an efficient orchestration schema to be operationalized as a cloud application mashup.

6.1 Identification of the Workflow Patterns

The first step is the identification of nodes and transitions of the WTS that compose one of the three workflow patterns discussed in Section 2. The procedure visits the graph and identifies when the structure presents a simple sequence of transitions, an exclusive-choice or a structured-loop. The procedure works by matching the graph with the patterns shown in Table 6.1. The precedence is the following: i) structured-loop, ii) exclusive-choice and finally iii) sequence.

Table 1 The identification of the resulting configuration is based on matching the three workflow patterns into the resulting WTS structure.

Workflow Pattern	WTS
structured-loop	
exclusive-choice	
sequence	

An example of identification is shown in Figure 16. Transitions are annotated with the corresponding capability used as a label for the arc.

The importance of establishing precedence is due to the reciprocal inclusion of the three patterns. Whereas the sequence is the most simple pattern to identify (see the left side of Figure 16), an exclusive-choice often contains also sequences (see the middle side of Figure 16). Furthermore, the structured-loop must include a choice (see the right side of Figure 16).

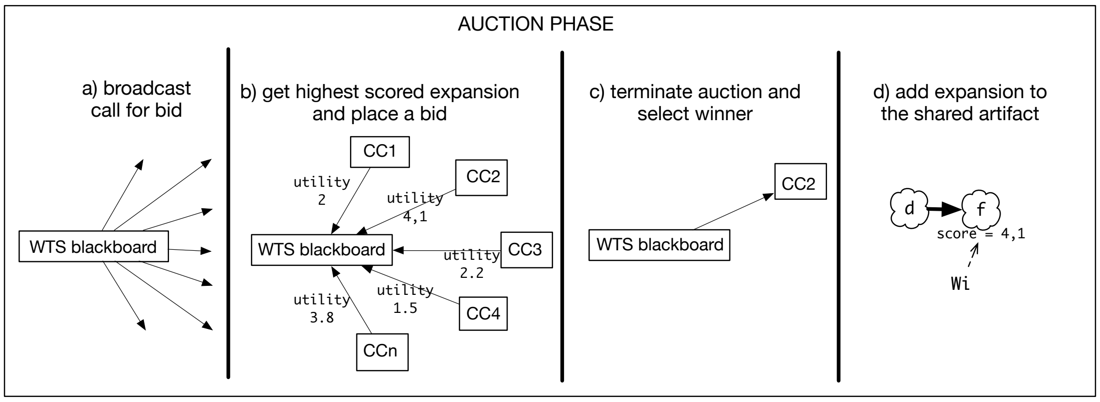

AUCTION PHASE
a) broadcast call for bid
WTS blackboard
b) get highest scored expansion and place a bid
CC1
CC2
CC3
CC4
CCn
utility 2
utility 4,1
utility 2.2
utility 1.5
utility 3.8
c) terminate auction and select winner
d) add expansion to the shared artifact
d
f
score = 4,1
Wi

Fig. 15 Steps of the Auction Cycle.

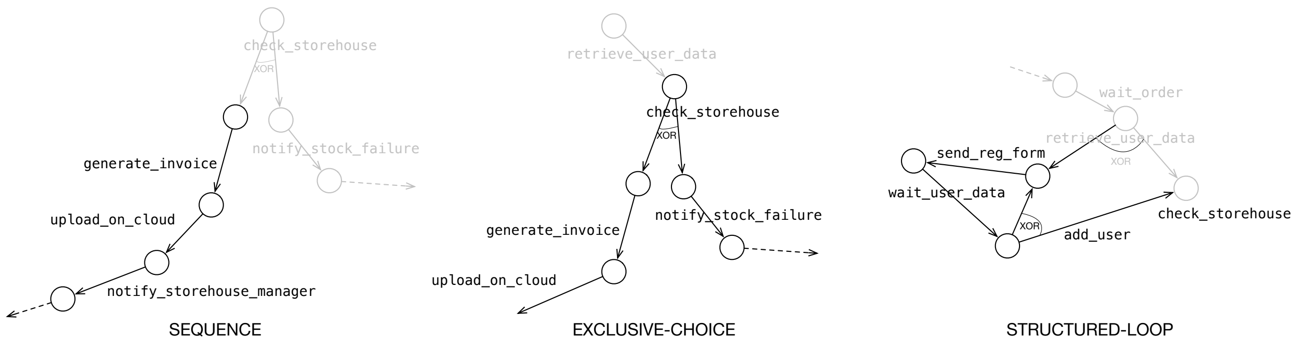


Fig. 16 Example of workflow patterns identified in the WTS: sequence, exclusive-choice and structured-loop.

6.2 Extracting Goal Operationalization

The second step is the identification of valid sub-graphs of the WTS, in which all the goals are satisfied.

The identification of nodes where goal satisfaction holds is executed during the WTS construction. Indeed, special states of the world, where a goal's TC or FS holds, are marked with labels (TC_holds_i and FS_holds_i respectively for the goal g_i). Figure 17 reports a portion of WTS generated for the fashion firm example. Transitions are annotated with the corresponding capability and nodes report the logic expressions that are true in the corresponding state of world (from W_3 to W_{12}). TC_i and FS_i respectively denote the triggering condition and final state of the goal g_i. So, for instance, the final state of g_1 and the trigger condition of $g4$ are satisfied in W_8. This Figure highlights the portion of the WTS where the satisfaction of g_2 (to_notify_invoice) holds.

The *goal_operationalization* procedure works as follows: given the goal g_i, it browses the WTS for identifying all the nodes in which respectively TC_i and FS_i are verified. Subsequently it exploits patterns identified in the previous step for building the sub-graph in which the capabilities may be used for addressing g_i. Aggregating the patterns (rather than single nodes) has the effect that the resulting sub-graph is robust with respect to both the exclusive-choice and the structured-loop patterns. Indeed, if the identified portion of the graph contains a choice or a loop, this structure is not truncated. For instance, in Figure 17, goal g_2 would be

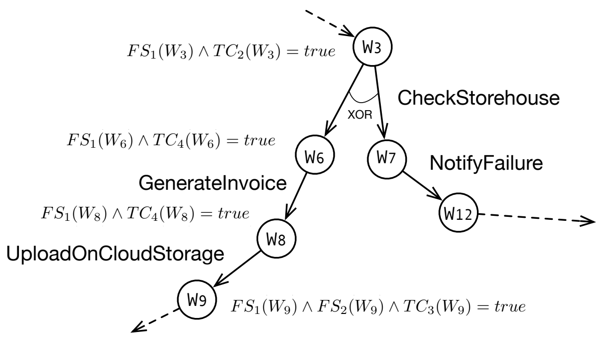

$FS_1(W_3) \wedge TC_2(W_3) = true$
W3
XOR
CheckStorehouse
$FS_1(W_6) \wedge TC_4(W_6) = true$
W6
W7
NotifyFailure
GenerateInvoice
$FS_1(W_8) \wedge TC_4(W_8) = true$
W12
W8
UploadOnCloudStorage
W9
$FS_1(W_9) \wedge FS_2(W_9) \wedge TC_3(W_9) = true$

Fig. 17 Example of navigation of the WTS for identifying suitable configurations.

satisfied by the sequence $(W_3, W_6, W_8, W9)$, however W_3 is involved in a exclusive-choice pattern, thus the resulting structure must also include the node W_7, for completing the exclusive-choice pattern.

6.3 The Resulting Orchestration Plans

The extracted configuration become executable when it is converted into an orchestration business plan. The main issue of this transformation is that a configuration is made of goals and capabilities, whereas an orchestration plan is a business process made of tasks, events, gateways and sequence/data/message flows. The transformation follows two principles:

– *Principle 1: Dynamic Association between Capability and Goals.* Each capability follows the lifecycle represented in Figure 18.
– *Principle 2: Distributed Control.* All the capabilities act autonomously (in parallel) and interact each other, with the aim of coordinating their behaviour. Interactions are driven by the need of exchanging data objects and by the external environment.

Principle 1 arises from the fact that each capability contributes to addressing one of the goals. A trigger-condition (TC) is associated with each goal, and it specifies when the goal may be pursued, and constitute the first condition that must hold for executing the related service. However, the capability also requires that a pre-condition must hold. At the same way, the post-condition reveals the success or failure of the service. Finally, the goal's final state (FS) asserts when the goal has been successfully addressed. Figure 18 summarises a schema of the typical flow of activities associated with a generic capability.

When several capabilities are requested for addressing the goal set then all the capability functional schemes must be merged.

Principle 2 states all the schemas must be combined to work concurrently (see Figure 19). This basically means generating a workflow in which each branch represents a different service (Figure 18). However, workflow patterns specified in the configuration regulate the way branches are composed. Different branches interact by two distinct synchronisation approaches.

Implicit mode: a branch waits until a condition of the state of the world is true. This state is generated as the final state of a prior service of the workflow. This model is specifically suitable for sequences of capabilities. It is implemented by means of a shared blackboard that stores the current state of the world.

Explicit mode: a branch requires to process something (a variable, a data object, a message) that is produced as output by another branch. In this case, a Query Interaction protocol [3] is employed to enable the direct exchange of data. This model is specifically suitable for implementing exclusive-choice and structured-loop patterns.

Figure 19 reports an exemplary slice of the workflow (using the BPMN 2.0 notation) obtained by applying the two transformation principles to the configuration 1 (shown in Figure 11). We highlight the branches corresponding the goal [to_notify_invoice] that involve three capabilities: *CheckStorehouse*, *GenerateInvoice* and *UploadOnCloudStorage*.
The first condition, $available(order) \wedge available(user)$, is the same for the three branches because it is the goal's triggering condition. For the sake of clarity, the workflow is optimized to avoid duplicate activities.

Therefore, when an order has been received from a registered user, the workflow waits for three possible events: i) an order data object to be processed, ii) a notification that the order has been accepted, or iii) an invoice to deliver. These are the entry points for the branches corresponding to the three capabilities to be executed: (`check_storehouse`, `generate_invoice`, `upload_on_cloud_storage`).

For example, after the service `check_storehouse` is executed, the capability's post-condition and the goal's final state are checked. An example of explicit synchronisation happens between the second and the third branches: the invoice data object is produced by the service *GenerateInvoice* and consumed by the *UploadOnCloudStorage*.

Figure 20 reports the aforementioned scenario with execution with some screenshots concerning the front-end parts of the exploited capabilities.

7 Related Work

The current state of the art in cloud computing delineates *mashup* as an innovative technology for the integration of cloud applications [22,1,18]. Compared to traditional 'developer-centric' composition technologies, a mashup is inspired by principles of flexibility and user-friendliness.

OpenCloudware [2] and FIWARE [12] represent examples of interesting initiatives that have begun implementing this vision through a backend infrastructure.

OpenCloudware [2] is a project coordinated by France Telecom Orange. It aims at building an open software engineering platform (PaaS) for the collaborative development of distributed applications to be deployed on multiple cloud infrastructures (IaaS). OpenCloudware supports mashup through a set of tools for managing

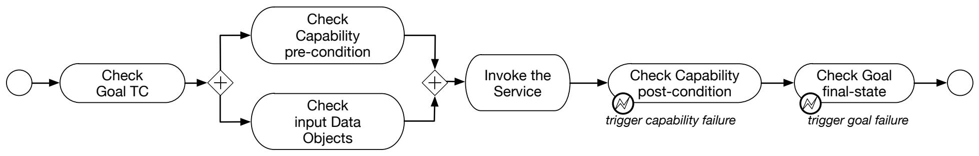


Fig. 18 Flow of activities corresponding to the execution schema related to a generic association (goal,capability).

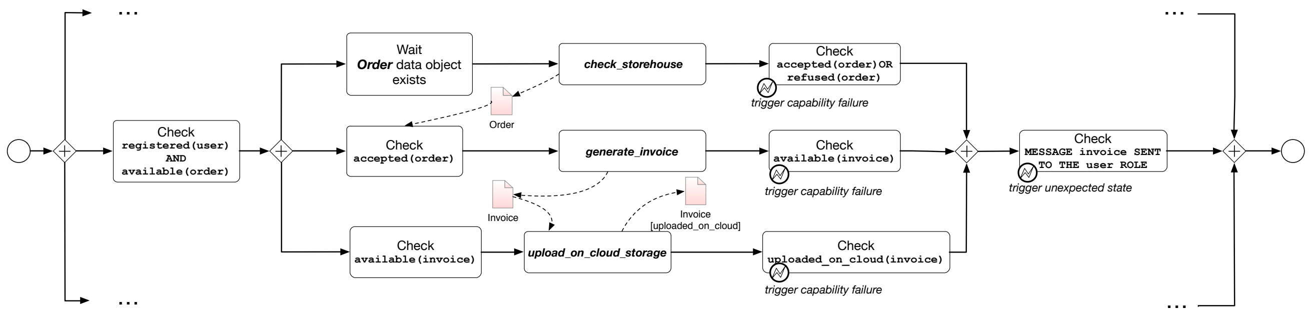


Fig. 19 Overview of the resulting workflow equivalent to a full-parallel execution model. Each branch of the workflow corresponds to a capability. It is built by instantiating and optimizing the lifecycle presented in Figure 18 with specific goal and capability properties.

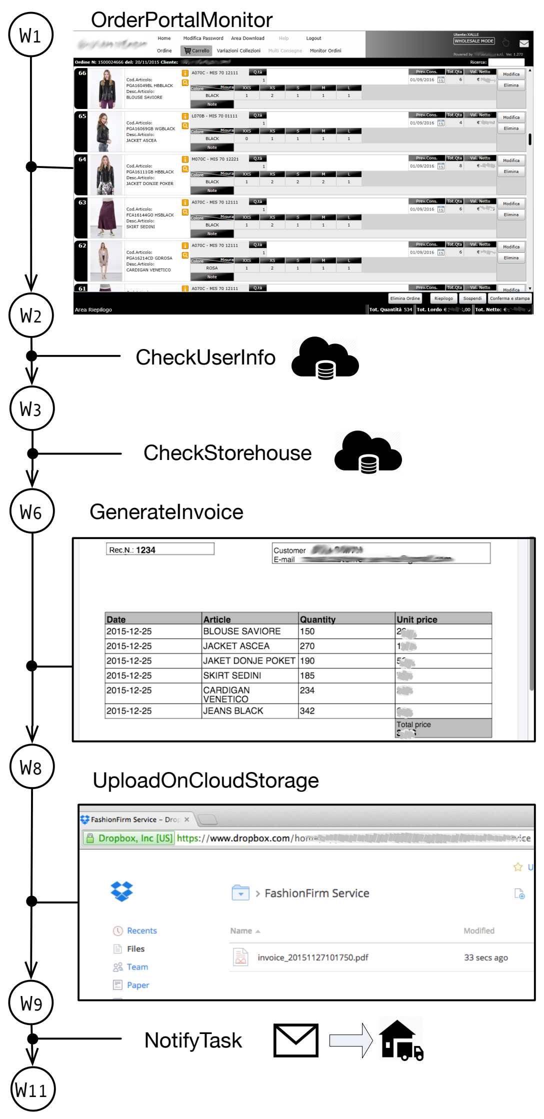
the lifecycle of such applications from many points of view: modelling, developing, deployment and orchestration. The composition of OpenCloudware services is operated through the definition of a Service Level Agreement. In [1] it is the responsibility of the system architect to define this contract. The authors provide a complex component, based on a MAPE-K loop [7], able to provide autonomic behaviour at the component level. Our approach extends this work because cloud capabilities are able of automatically generating the definition of their composition to be executed.

Conversely, FIWARE is an open architecture and a reference implementation of a service infrastructure [12] whose mission is: "to build an open, sustainable ecosystem around public, royalty-free and implementation-driven software platform standards that will ease the development of new Smart Applications (SaaS) in multiple sectors". FIWARE provides compelling software components, available through APIs, able to provide valuable Cloud platform functionalities. In particular, it offers an application mashup platform in which end users without programming skills can easily create web applications by manually integrating heterogeneous data, application logic, and UI components sourced from the Web. Generic Enablers are similar to our capabilities, but they lack the ability to interact autonomously. Composing them requires a deep knowledge of architecture and its API [10]. Integrating our approach with FIWARE's GE would reduce the time of development of the whole solution increasing modularity, scalability and flexibility of the final product.

Helin and Laukkanen [17] present an approach for composing workflows that are based on semantic type matching. As well as our approach, authors highlight the importance of ontology for creating semantically annotated services. The main difference is that their approach mainly automatizes finding and matching semantically similar web services, whereas the composition still requires the human intervention during the composition process.

In [6] authors model web service composition as a planning problem and use nondeterministic transition systems where a composition is achieved by model checking. Despite there exist similarities with our work, their strategy for building the transition system is not specifically suitable for running in a distributed fashion, a requirement that is necessary for the Cloud environment.

Another related work is Colombo [4] a framework for automatic web service composition that exploits relational database schema, atomic processes, message passing and a finite-state transition system. As well as our approach, they introduce the goal service, i.e. they make explicit that a composite service is aggregated for addressing a goal. The main difference is that in Colombo a service goal is directly represented as a transition system, that demands a user to learn very technical skills.



Rec.N.: 1234

Customer [redacted]
E-mail [redacted]

Date	Article	Quantity	Unit price
2015-12-25	BLOUSE SAVIORE	150	[redacted]
2015-12-25	JACKET ASCEA	270	[redacted]
2015-12-25	JAKET DONJE POKET	190	[redacted]
2015-12-25	SKIRT SEDINI	185	[redacted]
2015-12-25	CARDIGAN VENETICO	234	[redacted]
2015-12-25	JEANS BLACK	342	[redacted]
			Total price [redacted]

Fig. 20 Execution of a scenario for the fashion firm running example. Some screenshots of the cloud application mashup have been attached to the capabilities of configuration 1.

The goal-oriented approach for self-configuring cloud application mashups provides an alternative vision with respect to the classic workflow model definition. It aims at decoupling the technical skills for developing a service (how) from the analytic skill of describing mashup compositions (what). The result is an autonomic composition of services based on dynamic user's goals. The output is provided as a classical business process schema that a workflow engine is able to execute.

A different execution approach is followed by Blanchet et al. [5], who propose to view service orchestration as a conversation among intelligent agents, each one responsible for delivering the services of a participating organisation. An agent also recognises mismatches between its own workflow model and the models of other agents.

Another execution model is provided by OSIRIS [29], an Open Service Infrastructure for Reliable and Integrated process Support that consists of a peer-to-peer decentralised service execution engine and organises services into a self-organizing ring topology.

Conceptually close to our orchestration model, Hahn and Fischer, in [15] illustrate how a service choreography can easily be implemented through holons. An holon is a highly scalable and distributed organisation of autonomous entities based on a recursive approach to hierarchy. Their approach is a design-to-code technique based on model-driven transformations which result is a holonic multi-agent system. However, there are many works that study strategies for dynamic formation of holons for executing complex activities.

In the context of the SASSY research project [13], Gomaa and Hashimoto use software adaptation patterns for Service-Oriented Applications. The goal is not only to execute a mashup but to dynamically adapt distributed transactions at run-time. This is possible by separating the concerns of individual components of the architecture from concerns of dynamic adaptation. In their approach, the solution has been using a central manager that works as connector adaptation state-machine.

On the same direction, Ghezzi et al. [11] propose ADAM (ADAptive Model-driven execution) a mixed approach between model transformation techniques and probability theory. The modelling part consists of creating an annotated UML Activity diagram whose branches can have a probability assigned, plus an annotated implementation. Then an activity diagram becomes an MDP (Markov Decision Process). It is possible to calculate the possible values for the different executions and thus navigate the model in order to execute it.

A topic that is currently out of our work is the management of the QoS. Grassi et al. [14] propose a QoS-aware decentralised service assembly based on a dynamic set of agents may enter/leave the system, each offering a specific service. In this context, producing fully resolved assemblies is complicated by dependencies among service. Moreover, non-functional requirements and only the currently available services should be considered. Even further, all of this should be done using decentralised self-assembly (no external control, dynamic operation, no central control).

Concluding, in an open market paradigm, trust and reputation are frequently used to regulate social interactions and is to support decision-making when there is incomplete information. There are many works on service selection that incorporate trust in peer-to-peer networks. The novelty in [33] is a process for formulating trust as a belief in what a service will deliver based on evidence such as credential authority, experience, reputation and recommendation from multiple sources. Trust is then calculated as a function of beliefs, each of which may use different evidence types. So the problem is to create a trust model that allows for trust calculation.

8 Conclusions

In this paper, we describe our approach to the development of Cloud Application Mashup. This approach has been exploited in the context of a research project whose some characteristics have been presented as running example.

Our approach is based on the decoupling of what to do from how to do it. To this aim, we used a three levels architecture where the user specifies his problem and automatic tools build the resulting application.

On the developer side, services must be encapsulated within Cloud Capabilities, lively and autonomous SaaS applications that provide reasoning and composition facilities.

On the final user side, she has to define the business logic of the mashup in the form of goal-set as Goal-SPEC specifications. There is indeed the need of some minimal skill in order to specify the problem to be resolved. Goals must be specified adopting some ontological formality and conflict-free. In order to reduce the complexity of this work, off-line tools – such as that presented in [25] – may help the user in defining his desired application.

Service providers could have their own goals too. We are still working on integrating a third component close to goals and capabilities: norms. Norms are rules that must hold during all the phases of self-configuration but also during service orchestration and execution. To date, this is yet an ongoing work.

When both capabilities and goals are specified, then we can build a new mashup application just composing available cloud applications or web services exploiting the proactive characteristic of cloud capabilities as long as these exist in the repository. Capabilities can be figured out as proactive entities that bind an abstract description of some action to a real web service or cloud application. The main feature of cloud capabilities is to be 'social' i.e. able to interact in order to generate a shared solution to the set of user's goals.

The novelty of our approach lies in the fact that the user does not need to know how his mashup application will be composed or which components will be assembled. Each capability corresponds to specialised web services or cloud applications. Furthermore, redundant capabilities can make the resulting application safe from any service failure.

All these features are implemented in a middleware [26] that offers a whole architecture for monitoring goal injections, self-configuring ad-hoc solutions and finally to orchestrate Cloud components. The approach is not tied to a specific application domain. Indeed the specification of a domain ontology is a fundamental step for customising the middleware for the specific working context. Examples of different customizations can be found in the website[5]. To date, the middleware has been adopted for implementing a document sharing solution, a cloud mashup platform (the running example reported in this paper), a risk management system and a smart travel agency.

9 Acknowledgment

The research was partially funded by the Autonomous Region of Sicily, Project OCCP (Open Cloud Computing Platform), within the Regional Operative Plans (PO-FESR) of the EU Community.

References

1. T. Aubonnet, L. Henrio, S. Kessal, O. Kulankhina, F. Lemoine, E. Madelaine, C. Ruz, and N. Simoni. Management of service compositionbased on self-controlled components. *Journal of Internet Services and Applications*, 6(1):1–17, 2015.
2. T. Aubonnet and N. Simoni. Self-control cloud services. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 282–286. IEEE, 2014.
3. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Software-Practice and Experience*, 31(2):103–128, 2001.
4. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st international conference on Very large data bases*, pages 613–624. VLDB Endowment, 2005.
5. W. Blanchet, E. Stroulia, and R. Elio. Supporting adaptive web-service orchestration with an agent conversation framework. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 2005.
6. M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *ICAPS 2003 workshop on planning for web services*, pages 1636–1642, 2003.
7. B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap.* Springer, 2009.

[5] `http://aose.pa.icar.cnr.it/MUSA/`

8. M. Cossentino, D. Dalle Nogare, R. Giancarlo, C. Lodato, S. Lopes, P. Ribino, L. Sabatucci, and V. Seidita. Gimt: A tool for ontology and goal modeling in bdi multi-agent design. In *Workshop" Dagli Oggetti agli Agenti"*, 2014.
9. E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995(1072):5, 1990.
10. M. Fazio, A. Celesti, F. G. Márquez, A. Glikson, and M. Villari. Exploiting the fiware cloud platform to develop a remote patient monitoring system. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 264–270. IEEE, 2015.
11. C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 33–42. IEEE Press, 2013.
12. A. Glikson. Fi-ware: Core platform for future internet applications. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.
13. H. Gomaa and K. Hashimoto. Dynamic self-adaptation for distributed service-oriented transactions. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 11–20, 2012.
14. V. Grassi, M. Marzolla, and R. Mirandola. Qos-aware fully decentralized service assembly. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 53–62. IEEE Press, 2013.
15. C. Hahn and K. Fischer. Service composition in holonic multiagent systems: Model-driven choreography and orchestration. In *Holonic and Multi-Agent Systems for Manufacturing*, pages 47–58. Springer, 2007.
16. V. Krishna. *Auction theory*. Academic press, 2009.
17. M. Laukkanen and H. Helin. Composing workflows of semantic web services. In *Extending Web Services Technologies*, pages 209–228. Springer, 2004.
18. S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud computing—the business perspective. *Decision support systems*, 51(1):176–189, 2011.
19. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.
20. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(4):333–351, 2003.
21. A. Newell. The knowledge level. *Artificial intelligence*, 18(1):87–127, 1982.
22. M. P. Papazoglou and W.-J. van den Heuvel. Blueprinting the cloud. *IEEE Internet Computing*, 6:74–79, 2011.
23. M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, volume 1, pages 479–486, 2001.
24. L. Sabatucci and M. Cossentino. From Means-End Analysis to Proactive Means-End Reasoning. In *Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Florence, Italy*, May 18-19 2015.
25. L. Sabatucci, C. Lodato, S. Lopes, and M. Cossentino. Towards self-adaptation and evolution in business process. In *AIBP@ AI* IA*, pages 1–10. Citeseer, 2013.
26. L. Sabatucci, C. Lodato, S. Lopes, and M. Cossentino. Highly customizable service composition and orchestration. In S. Dustdar, F. Leymann, and M. Villari, editors, *Service Oriented and Cloud Computing*, volume 9306 of *Lecture Notes in Computer Science*, pages 156–170. Springer International Publishing, 2015.
27. L. Sabatucci, P. Ribino, C. Lodato, S. Lopes, and M. Cossentino. Goalspec: A goal specification language supporting adaptivity and evolution. In *Engineering Multi-Agent Systems*, pages 235–254. Springer, 2013.
28. R. Siebeck, T. Janner, C. Schroth, V. Hoyer, W. Wörndl, and F. Urmetzer. Cloud-based enterprise mashup integration services for b2b scenarios. In *Proceedings of the 2nd workshop on mashups, enterprise mashups and lightweight composition on the web, Madrid*, 2009.
29. N. Stojnic and H. Schuldt. Osiris-sr: A safety ring for self-healing distributed composite service execution. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 21–26, 2012.
30. D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM, 2008.
31. W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.
32. M. J. Wooldridge. *Reasoning about rational agents*. MIT press, 2000.
33. C. H. Yew and H. Lutfiyya. A middleware and algorithms for trust calculation from multiple evidence sources. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 83–88, 2012.
34. E. Yu and J. Mylopoulos. Why goal-oriented requirements engineering. *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, 15, 1998.
35. J. L. Zhao, M. Tanniru, and L.-J. Zhang. Services computing as the foundation of enterprise agility: Overview of recent advances and introduction to the special issue. *Information Systems Frontiers*, 9(1):1–8, 2007.