A Goal-Oriented Approach for Self-Configuring Mashup of Cloud Applications

Luca Sabatucci*, Salvatore Lopes* and Massimo Cossentino*
* ICAR-CNR, Palermo, Italy
Email: {sabatucci, s.lopes, cossentino}@pa.icar.it

Abstract—This paper presents a general approach for automatic composing mashups of applications distributed over the cloud. The approach implies to wrap existing services in smart and autonomic entities, namely cloud capabilities. These are able to interact and coordinate themselves in order to establish different ways to orchestrate their contained services. The main enabler of this technology is based on an explicit distinction between user’s goals and the way to address them. A couple of language has been adopted to describe respectively the mashup logic in terms of goals and the available functionalities in terms of capabilities. A running example has been developed for extending B2B business processes of a fashion enterprise.

I. INTRODUCTION

The market turbulence of recent years is an indication of a new phase of globalization, one in which the ability to satisfy customer expectations in a quickly changing environment is the core differentiator for enterprises [1]. Companies require ways to make their processes more flexible opening, at the same time, their business process to the direct access of users.

Technology can play an important supporting role in enabling organizations to become more agile. In particular, Cloud computing focuses on maximizing the effectiveness of shared resources and information (that are provided to users on-demand) reducing the overall cost by using less power, air conditioning, rack space, etc.

Despite the fact that Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access, Cloud applications are currently developed as monolithic solutions tethered to proprietary stack architectures in which the provider typically runs all elements of the service [2]. These architectures may represent a barrier for third-part developers to mix and match services freely from diverse cloud service levels to configure them dynamically to address application needs [2].

The objective of Cloud Application Mashup is to enable easier customization and composition of SaaS applications from several providers by providing a cohesive solution that offers improved functionality to clients. The mashup task involves complex protocols with an automatic data and process integration, and preserves global application consistency.

So far, mashup is mainly intended as an instrument for web developers for integrating content from more than one source in a new single graphical interface. A representative example is HousingMaps (http://www.housingmaps.com), which combines rental listings from a popular advertisements website with Google Maps for providing a visual representation of local apartments for rent. Other common examples are online magazines that include buttons for social sharing (for instance: The New York Times (http://www.nytimes.com), CNN.com (www.cnn.com), BBC (http://www.bbc.com)).

Now we are moving towards a variety of web service that run over the Cloud as SaaS. Notable examples are: Google Applications, Dropbox, Microsoft Office365, the mashup technology is evolving for mixing processes together with data and user interfaces.

This paper presents an approach for automatically generating Cloud Application Mashup. The vision is that Mashups are created for the consuming user, often directly by the users themselves, so that they can take advantage of software licensing and billing model based on the pay-per-use concept. Therefore users will establish or modify situational collaborations for integrating services from a variety of cloud providers. Cloud services should be available on public cloud marketplace in which providers store their offerings. Clients can discover and buy the needed services to use for their own mashup. Therefore, a mashup self-composition engine should act as a run-time mediator between user’s goals and atomic cloud services in order to realize the desired mashup application.

In recent works we have adopted GoalSPEC [9], a goal-oriented [3] language in order to describe the expected behavior of a complex distributed system in terms of states to be addressed. The algorithm for automatically selecting and aggregating services in order to address user’s goals has been presented in [11]. However the main limit of that algorithm relies on the centralized logic and in particular it is the difficulty in scaling up for an increasing number of goals and services.

The main contribution of the paper is a three-layer architecture for implementing Cloud Capability and supporting cloud application mashup. A cloud capability is a smart and autonomous container of traditional web services. It is a lively application, running at SaaS level, inspired by the principle of autonomic computing: sensing the environment, making proactive decisions and interacting with other cloud capabilities in order to organize a coordinated behavior. To enable the automatic composition of services, these must be available together with relevant aspects for their integration and usage through ad-hoc description languages. We adopted a capability description language that exploits predicate logic in order to specify when and how to use a service. Decoupling the
specification of what the system has to do (user’s goals) from how it will be done (capabilities) allows the self-configuration engine to compose the expected behavior on the occurrence.

As a running example we report the case of cloud mashup built for a fashion enterprise. The B2B process is described through a set of goals that are introduced into the system thus to become a stimulus for self-configuring ad-hoc solution.

The marketplace has been instrumented with a sufficient set of services for addressing the request. Therefore, by exploiting their self-configuration protocols, it is responsibility of the corresponding cloud capabilities to configure the expected B2B mashup.

We envisage this approach could have substantial impact for IT by improving the return-on-assets of the existing systems. 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 [4]. Short and cheap development cycles make B2B solutions available for small and medium enterprises.

The paper is organized as follows: Section II presents the main concepts of the approach: state, goal, capability and configuration. Section III presents a real scenario occurred during a research project and introduces the elements for enabling the self-configuration. Section IV illustrates details of the self-configuration module: the three layer-architecture and the cloud capability. Section V focuses on the self-configuration distributed process, whereas Section VI explains the dynamic workflow generation related to a mashup. Related works are discussed in Section VII whereas conclusions are given in Section VIII.

II. PRELIMINARY DEFINITIONS AND CONCEPTS

In this subsection we introduce some of the main concepts that are used along the paper.

A State of the World is an abstraction made to let the system reasoning at the knowledge level [5]. It arises from the consideration that a software system has (partial) knowledge about the environment in which it runs. The classic way for expressing this property is \( \langle \text{Bel} \ a \ \varphi \rangle \) [6] that specifies that a software agent \( a \) believes \( \varphi \) is true, where \( \varphi \) is a fact that describes a generic state of affair. This is similar to the concept of fluent found in situation calculus. The definition of relevant and coherent states of the world requires a preliminary analysis of the domain that is perfectly captured in an ontology describing classes, properties and individuals of interest. The proposed approach is independent on the specific instrument chosen to model the domain (e.g. OWL-S [7], POD [8]).

Definition 1 (State of the World): The state of the world in a given time \( t \) is a set \( W_t \subset S \) where \( S \) is the set of all the non-negated facts \( s_1, s_2 \ldots s_n \) that can be used in a given domain.

\( W_t \) has the following characteristics:

\[
W_t = \{ s_i \in S | (\text{Bel} \ a \ s_i) \}
\]  

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

A Capability is a self-contained process, which can access and modify the current state of the world. These are used as wrappers for coordinating existing cloud applications and services. The capability also has the advantage of being composable in order to address a complex result.

Definition 2 (Capability): A capability \( \langle \text{evo, pre, post} \rangle \) is an atomic and self-contained action the system may intentionally use to address a given evolution of the state of the world. The evolution, denoted as \( \text{evo} : W \rightarrow W \) 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 \( W_{t+1} \) by manipulating statements in \( W_t \). The capability may be executed only when a given pre-condition is true \( (\text{pre}(W_t) = \text{true}) \). Moreover, the post-condition is a run-time helper to check if the capability has been successfully executed \( (\text{post}(W_{t+1}) = \text{true}) \).

The concept of Goal is often used in the context of business processes for representing enterprise strategic interests that motivate the execution of a specific workflow [3]. It is “a desired change in the state of the world an actor wants to achieve”.

Definition 3 (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

the goal is addressed iff \( tc(W_t) \land fs(W_{t+k}) \) where \( k > 0 \) \n
i.e. a goal is addressed if and only if, given the trigger condition is true, then the final state must be eventually hold true somewhere on the subsequent temporal line.

The user should specify his mashup application by means of a goal-set.

A Configuration is defined as a set of tuples of type \( \langle g, h \rangle \) that fully address the goal-set specified by the user, where for each goal \( g \) of the goal-set, \( h \) is a simple or composed capability selected for addressing it. In other words a configuration associates each goal of the goal-set with a simple or composed capability. The main property of a configuration is that executing its capabilities will address the whole goal set. Section VI provides details about how to convert a configuration into an executable workflow.

Section V illustrate the central part of the paper, i.e. how system may generate configurations as response to user’s goals.

III. AN EXAMPLE OF CLOUD APPLICATION MASHUP

This section provides an overview of the self-configuration process by means of a running example extracted from the activities conducted in the OCCP research project\(^1\).

\(^1\)The OCCP project web-page is available at http://aose.pa.icar.cnr.it/OCCP/
A. 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. In order to enlarge 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 (OrderPortal) from the other side. SWHouse is demanded also to enrich the FashionFirm business process by adding new services for customer management.

These new services are conceived as mashup of cloud application with the aim to improve the costs-benefit ratio. In fact, this allows 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 will be used 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 RetailStore requests for a product stock through the OrderPortal, the system merges the legacy services with external applications provided by cloud computing providers: a Cloud Storage system is used for storing and delivering receipts to RetailStore, Voicemail for communicating the delivery status and, finally, a Cloud Calendar service for annotating the delivery status.

B. Elements for Self-Configuration

The domain expert has 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,
- unary user’s properties: registered, has_cloud_space
- unary document’s properties: uploaded_on_cloud
- unary order’s properties: available, accepted, refused
- binary properties: notified(document,user), mailed_perm_link(document,user) is-a notified

For instance the class user denotes FashionFirm’s users, whereas the class order denotes a complex description of items and quantity that a user wants to buy from FashionFirm.

The developer uses the ontological description of the scenario for defining the desired mashup by means 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 [9]. The corresponding GoalSPEC definition for the running example is reported in the following list:

```
GOAL to_wait_order:
  WHEN MESSAGE order RECEIVED FROM THE user ROLE
  THE SYSTEM SHALL ADDRESS available(order)
GOAL to_notify_invoice:
  WHEN available(user) AND available(invoice)
  THE SYSTEM SHALL ADDRESS
  MESSAGE invoice SENT TO THE user ROLE
GOAL to_deliver_order:
  WHEN MESSAGE invoice SENT TO THE user ROLE
  THE SYSTEM SHALL ADDRESS
  MESSAGE deliveryorder SENT TO THE storehouse_manager ROLE
GOAL to_notify_failure:
  WHEN available(order) AND refused(order)
  THE SYSTEM SHALL ADDRESS
  MESSAGE failure SENT TO THE user ROLE
```

In the above listing, all the words in upper case are keywords of GOALSpec language whereas the word 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 clear that there must be a cloud application or a combination of cloud applications that effectively satisfies each single goal. In order to automatically select and bind services to user’s goals, the self-configuration requires additional information about available services. This meta-data is provided by means of the capability language.

First, the order must be processed via the eCommerce website (order_portal). The OrderPortalMonitor Capability is responsible of hosting the website and to wait for new orders from users.
A screenshot of the eCommerce website managed by this capability is shown in Figure 1.

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

**CAPABILITY CheckStoreHouse:**
- **Pre-Condition:** available(order)
- **Post-Condition:** accepted(order) OR refused(order)
- **Evolution:** [add(accepted(order)), add(refused(order))]

If the requested products are available, the corresponding invoice should be delivered through the **UploadOnCloudStorage**. In alternative, the invoice is stored locally and a link will be communicated to the user via mail. Figures 2 and 3 show respectively the **UploadOnCloudStorage** working with the commercial Dropbox API\(^2\) and the **ShareFileLink** that uses Google Drive services\(^3\) to send the notification.

**CAPABILITY UploadOnCloudStorage:**
- **Pre-Condition:** available(invoice)
- **Post-Condition:** uploaded_on_cloud(invoice)
- **Evolution:** [add(uploaded_on_cloud(document))]

**CAPABILITY ShareFileLink:**
- **Pre-Condition:** uploaded_on_cloud(document)
- **Post-Condition:** mailed_perm_link(document, email)
- **Evolution:** [add(mailed_perm_link(document, email)), add(sent(document, user))]

It is worth to detail the link between capabilities and web services. During the self-configuration phase all the available capabilities in the repository are automatically checked against their compatibility to the requested goal. The way the automatic self-configuration happens is reported in Section V. The chosen capabilities are linked to the corresponding web services during the orchestration phase (Section VI).

### C. Resulting Configurations

The self-configuration phase consists in building the state transition system by exploiting available capabilities. A state transition system is a common model used to formally depict the behavior of discrete systems. The self-configuration phase manages the dynamic binding between goals and capabilities by building a state transition system. This transition system is exploited for configuring a suitable system behavior for the assigned goal-set.

A relevant part of the transition system for the B2B scenario is reported in Figure 4. This graph’s nodes represent possible (stable) states of the system. The transition from a state to another state is either due to an external event (denoted with <=) or due to the execution of a web service (internal event, denoted with !). Initially the system is in a state in which it waits for user orders from the order_portal service. When an order incomes then the system retrieves user’s data from a database. Here, if the user is not already registered then the system initializes a registration procedure (send_reg_form and register_user services), otherwise the order is checked over the storehouse database (check_storehouse service) thus to be processed until the ordered products are ready to be delivered (notify_storehouse_manager service).

An interesting part of Figure 4 is that in which the invoice is ready to be sent (after the generate_invoice service).

---

\(^2\)https://www.dropbox.com/developers-v1/core/docs

\(^3\)https://developers.google.com/drive/v2/reference/
and the system must send it to the user. Here, combining the available capabilities, the system has two choices: either to directly upload the file on the user’s cloud storage (upload_on_cloud service) or to locally store the file and to send a mail containing the file’s url (obtained by combining the upload_on_cloud and the share_link_file services).

We highlighted this situation because in points like this the system is able to generate different behaviors for addressing the same goals. Configurations capture this kind of variability that is fundamental to customize the final system.

For instance, the self-configuration phase for the running example results in three different configurations, each of them resolving the order management process by employing a different set of capabilities. These are extracted as subset of the reported graph and listed below.

Configuration 1

to_wait_order <-- !wait_order,
!retrieve_user_data
to_notify_invoice <-- !check_storehouse,
!generate_invoice,
!upload_on_cloud
to_deliver_order <-- !notify_storehouse_manager
to_notify_failure <-- !notify_stock_failure

Configuration 2

to_wait_order <-- !wait_order,
!retrieve_user_data,
!send_reg_form,
!wait_user_data
to_notify_invoice <-- !check_storehouse,
!generate_invoice,
!upload_on_cloud
to_deliver_order <-- !notify_storehouse_manager
to_notify_failure <-- !notify_stock_failure

Configuration 3

to_wait_order <-- !wait_order,
!retrieve_user_data
to_notify_invoice <-- !check_storehouse,
!generate_invoice,
!upload_on_cloud,
!share_file_link
to_deliver_order <-- !notify_storehouse_manager
to_notify_failure <-- !notify_stock_failure

IV. The Proposed Model for Self-Configuration

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 global functioning with respect to defined user’s goals [10].

A. A Three-Layered Architecture for Self-Configuration

The proposed approach is structured in three inter-operating 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 behavior of the system in terms of 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 interpreted and analyzed and therefore trigger the need of 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 [11]. It aims at selecting the capabilities (and configuring them) as a response to requests defined at the top layer. This 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 task and 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 VI 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.

B. Implementing the Cloud Capabilities

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

Whereas traditionally services and cloud applications are passive entities that act when receive the control [7], [12], cloud capabilities are lively cloud applications (running at SaaS) inspired to the principles of autonomic computing. Each cloud capability keeps its own control, it is able of sensing the surrounding environment, making proactive decisions, and interacting with other capabilities for organizing a common behavior [11], [13].

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 5: a general-purpose reusable core and a service customized 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.

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 to specify 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 II).

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 aforementioned capability.

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(oauth2)
).

The first compartment (semantic description) contains exactly information already discussed in Section III-B: 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 latter 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).

V. SELF-CONFIGURING A MASHUP

This section focuses on the central 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 [9] and [13] 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 sequence of capabilities 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 in the corresponding capability’s property evolution. This describes the expected changes in terms of 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 sequence of capabilities produces a multi-step evolution of the state of the world i.e $e = \{W_1, W_2, \ldots, W_n\}$. Such 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: $W_{k2} \in e : k2 \geq k1$.

A. The Proposed Strategy for Self-Configuration

The possible evolution paths of a system are modeled as a state transition system (Figure 4) where nodes are states of the world and transitions are due to the execution of capabilities. We refer to this structure with the name of world transition system (WTS, hereafter). The WTS is built with the contribution of a subset 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, in order to avoid the concurrent modification of the same WTS, cloud capabilities enforce a blind auction protocol [14] 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 that are privately stored. These new states are evaluated with respect to the goals, according to a global scoring function. This is used for predicting how much the new state is promising with respect to the goal-set.

Periodically, when a new call-for-bid incomes, each cloud capability selects the most promising state it has generated during past expand-and-evaluate iterations. The state score is used for setting a bid for participating to the auction. Who wins the auction gains the permission to update the WTS.
This strategy rewards those capabilities that promise to improve the WTS by increasing the global goal satisfaction.

B. The Expand-and-Evaluate Cycle

When a user specifies a goal-set to address, cloud capabilities enters in an expand-and-evaluate cycle, working with the WTS in read-only access. The cycle is described in the following.

1) The 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 6.a);
3) The capability simulates the effects of its wrapped service by generating new states through the evolution property (Figure 6.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 6.c); a node is also marked as $TC_{\text{holds}}$, or $FS_{\text{holds}}$, if the node satisfies respectively the triggering condition $TC_i$ or the final state $FS_i$ of the goal $g_i$.

C. The Global Scoring Function

The aim of the scoring function is to predict how much a state of the world is near to the final state where a goal is satisfied. The principle is that a state of the world is described by a set of numerable statements. We defined a function that evaluates the potential impact of each of these statements to addressing a goal. To the aim of producing a quantitative measurement, the function rewards statements that provide a positive impact to a goal and it penalizes statements that do not provide a positive impact to a goal.

The function is defined as follows:

$$score(W) = \sum_{g_i} \frac{1 + num_{\text{rel-stats}}(W, g_i)}{num_{\text{stats}}(W)}$$

where, given a state $W$, $num_{\text{rel-stats}}(W, g_i)$ is defined as the number of statements contained in $W \cap (TC_g \cup FS_g)$, whereas $num_{\text{stats}}(W)$ is the cardinality of $W$, i.e. the number of statements contained in $W$. For instance, if $W = \{s_1, s_2, s_4, s_5\}$ and $g = \langle s_2 \land s_4, s_4 \lor s_5 \rangle$ then $num_{\text{stats}} = 5$ and $num_{\text{rel-stats}} = 3$ because $\{s_2, s_4, s_5\}$ are relevant for $g$.

Figure 7 illustrates Function 3 plotted as a stacked line chart for highlighting the score trends. Setting to constant the $num_{\text{stats}}$ in the formula, the score is higher the more the statements are relevant for the goal satisfaction. Conversely, making the $num_{\text{rel-stats}}$ 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 quickly converge to exactly the desired final state. Clearly the prediction based on this heuristic is not guaranteed to be neither optimal or perfect, but empirically it revealed sufficient for the specific purpose of speeding the exploration of the graph.

D. The Auction Cycle

The auctioneer launches a new blind auction with a constant time interval. The auction starts when a call-for-bid is sent to the capabilities. They have a fixed deadline to reply with a bid.

Each participant selects the node –from its private expand list– with the highest utility, calculated as follows: the node’s score plus the number of $TC_{\text{holds}}$ and $FS_{\text{holds}}$ for that state.
The highest utility is the bid to send back to the auctioneer (see Figure 6).

There is not counter-offer, the auction closes when all capabilities replied or at a predefined deadline. The capability with the highest bid wins the auction and therefore it updates the WTS consequently. In practice, it copies the selected node from its private data structure to the global WTS, also reporting eventual transitions with pre-existent nodes. For instance, Figure 6 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 number of times (expanding the WTS is no more possible).

Figure 8 reports an example of WTS generated for the fashion firm running example where the conditions hold:

\[
\begin{align*}
TC_1(W_1) &= \text{true} \\
FS_1(W_2) &= \text{true} \\
FS_1(W_3) \land TC_2(W_3) &= \text{true} \\
FS_1(W_4) \land TC_2(W_4) &= \text{true} \\
FS_1(W_5) \land TC_2(W_5) &= \text{true} \\
FS_1(W_6) \land TC_4(W_6) &= \text{true} \\
FS_1(W_7) \land FS_1(W_7) &= \text{true} \\
FS_1(W_8) \land TC_4(W_8) &= \text{true} \\
FS_1(W_9) \land FS_2(W_9) \land TC_3(W_9) &= \text{true} \\
FS_1(W_{10}) \land TC_4(W_{10}) &= \text{true} \\
FS_1(W_{11}) \land FS_2(W_{11}) \land FS_3(W_{11}) &= \text{true}
\end{align*}
\]

The transformation follows two principles:

1. **Principle 1: Dynamic Association between Capability and Goals.** Each capability – selected for addressing a goal – will be executed according to the lifecycle represented in Figure 10.

2. **Principle 2: Distributed Control.** All the capabilities act autonomously (in parallel) and interact to the aim of coordinating their behavior. Interactions are driven by the current state of the world and by the need of exchanging some data objects.

The Principle 1 arises from the fact that each capability

\[\text{conf1, conf2, conf3} \ldots\]

\[\text{wait_order, retrieve_user_data, send_reg_form, wait_user_data, print_invoice, check_storehouse, generate_invoice, upload_on_cloud, share_file_link, notify_storehouse_manager, notify_stock_failure, supervise_document} \ldots\]

\[\text{Fig. 9. Example of set of configurations.}\]
Fig. 10. Flow of activities corresponding to the execution schema related to a generic association (goal,capability).

Fig. 11. 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 10 with specific goal and capability properties.

contributes to address one of the goals. A triggering-condition (TC) is associated to each goal and specifies when the goal becomes ready. This is the first condition that must hold for executing the related service. However, the capability also requires 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 10 summarizes a schema of the typical flow of activities associated to 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 schemes must be combined through a parallel gateway (see Figure 11) thus generating a workflow in which each branch represents a different service (Figure 10). Different branches interact by two different synchronization approaches. Implicit mode: a branch waits until a condition of the state of the world is true. This state is generally generated as the final state of a prior service of the workflow. This model is implemented by means of a shared blackboard that stores the current state of the world. Explicit mode: a branch requires to process some data object that is produced as output by another branch. In this case a Query Interaction protocol [15] is employed to enable the direct exchange of data.

Figure 11 reports an exemplar slice of the workflow (using the BPMN 2.0 notation) obtained by applying the two transformation principles to the configuration 1 (shown in Figure 9). We highlight the branches corresponding the [to_notify_invoice] goal that involves three capabilities: CheckStorehouse, GenerateInvoice and UploadOnCloudStorage. The first condition available(order) ∧ available(user) is the same for the three branches because it is the goal’s triggering condition. For the sake of clarity the workflow has been simplified to avoid duplicate activities, as shown in Figure 11.

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 that execute check_storehouse, generate_invoice and upload_on_cloud_storage respectively.

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

Figure 12 reports the aforementioned scenario of execution with some screenshots concerning the front-end parts of the exploited capabilities.

VII. RELATED WORK

The current state of the art in cloud computing delineates mashup as an innovative technology for the integration of cloud applications [2], [16], [17]. Compared to traditional ‘developer-centric’ composition technologies, mashup is inspired to principles of flexibility and user-friendliness.

OpenCloudware [18] and FIWARE [19] represent a couple of worldwide initiatives that have began implementing this vision through a backend infrastructure.

OpenCloudware [18] is a project coordinated by France Telecom Orange. It aims at building an open software engi-
neering platform (PaaS) for the collaborative development of distributed applications to be deployed on multiple cloud infrastructures (IaaS). OpenCloudware support mashup through a set of tools to manage the lifecycle of such applications from many point of views: modelling, developing, deployment and orchestration.

Conversely, FIWARE is an open architecture and a reference implementation of a service infrastructure [19] 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”. It offers an application mashup platform allowing end users without programming skills to easily create web applications by manually integrating heterogeneous data, application logic, and UI components sourced from the Web.

Our approach for self-configuring mashups provides an alternative vision respect to the classic workflow model definition. It aims at decoupling the technical skills for developing a service from the analytic skill of describing mashup compositions.

Helin and Laukkanen [20] present an approach for composing workflows that is 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 [21] authors model web service composition as a planning problem and use non-deterministic transition systems where 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, requirement that is necessary in the Cloud environment.

Another related work is Colombo [22] 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.

In [16] authors explicitly focus on the composition of cloud components and highlight the need for self-controlled service components. They adopt a MAPE-K loop [23] to provide autonomic behavior at component level. However the composition is still modeled at design-time by developers.

**VIII. 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 specify 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 GoalSPEC 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 [24] – may help the user in defining his desired application.

Service providers could have their own goals too. We are still working at 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 specialized 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 [13] 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 customizing the middleware for the specific working context. Examples of different customizations can be found in the website5. 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.

IX. 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


5http://aose.pa.icar.cnr.it/MUSA/