Self-Configuring Mashup of Cloud Applications

A.Cavaleri, M. Cossentino, C. Lodato, S. Lopes, L. Sabatucci
ICAR-CNR, Palermo, Italy
Email: {sabatucci,s.lopes,cossentino}@pa.icar.cnr.it

Abstract—**This paper presents a general approach for automatic composing and orchestrating applications distributed over the cloud. The process is driven by user requirements that are made explicit though a goal specification language. The self-configuration module dynamically organizes a mashup application by composing existing cloud application as atomic brick to compose. Finally the orchestrator module is responsible of a seamless enacting of the selected cloud applications. A prototype has been implemented as a multi agent system for implementing the business process for a company working for fashion firms.**

I. Introduction

Cloud computing focuses on maximizing the effectiveness of shared resources and information (that are provided to users on-demand) and reducing the overall cost by using less power, air conditioning, rack space, etc. Cloud applications are currently developed as monolithic solutions tethered to proprietary stack architectures in which the provider typically runs all elements of the service [9]. These architectures are a barrier for third-part developers to mix and match services freely from diverse cloud service tiers to configure them dynamically to address application needs [9].

The objective of Cloud Application Mashup is to enable easy customization and composition of SaaS applications from many providers by providing a cohesive solution that offers improved functionality to the client. In this paper we report a practical experience of Cloud Application Mashup that derives from a background of research in self-configuring system and dynamic workflow composition.

We describe an architecture for closing the gap between goal-oriented requirement engineering [20] and autonomous systems [5] with the aim of creating a highly customizable orchestration of distributed services. The request for a mashup is based on a technique we called goal-injection: a goal is the high-level specification of the kind of service desired by the user. Goal orientation is used for decoupling the specification of what the system has to do from how it will be done. Service compositions must not be programmed. Once a goal has been introduced into the system, it becomes a stimulus for self-configuring ad-hoc solutions in order to fulfill the request. The basic assumption is to make available some wrappers for existing services and cloud applications, namely capabilities. It is a responsibility of the system to aggregate capabilities thus to obtain composed behavior. Finally, orchestration is based on establishing (temporary) collaborations between services according to self-configured conditions. A prototype has been developed as a multi-agent system for implementing a customer service business process of the B2B process of a big fashion firm[1].

The paper is organized as follows: Section II presents the concept of Cloud Application Mashup and provides a real scenario emerged during a research project. Section III introduces the goal-oriented language to specify user's requirements. Section IV describes a three layer architecture for self-configuration that is the core for service mashup. Section V presents the orchestrator module and describes the ad hoc business process obtained at run-time. Related works are discussed in Section VI whereas some Conclusions are given in Section VII.

II. Cloud Application Mashup

The term *mashup* is a shortcoming for applications created by integrating modular components. It originates in the context of web applications for referring to a some kind of collaboration between websites and/or webservices. In general a web-based mashup is a resource that combines existing sources, whether it is content, data or application functionality, from more than one resource enabling end users to create and adapt individual information centric and situational applications.

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. Enablers of web mashup are i) popular standard-based interface/communication technologies (such as WSDL, REST, and RSS) and ii) the fact many Internet companies opened up their data to be used through a set of APIs.

Now, we are moving towards a variety of Web Applications that run over the Cloud as SaaS, the new mashup trend is mixing data but also processes.

A. Cloud Application Mashup: The Enterprise Vision

Cloud Applications are currently developed as monolithic solutions in which the SaaS layer is tethered to proprietary cloud stack architecture. The cloud provider runs all elements of the service and presents a complete application to the client [9]. This architecture hinders third-part developers to mix and match services freely from diverse cloud service tiers to configure them dynamically to address application needs [9].

Cloud Application Mashup objective is to surpass vendor lock-in (i.e. the situation in which customers are dependent

[1]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.

on a single manufacturer or supplier for some product) in order to customize and combine SaaS applications from many providers. It provides ways to orchestrate a cohesive solution that assembles virtual services in order to offer improved functionality to the client.

We refined the current vision of Cloud Application Mashup as follows:

- Cloud services will be available via a cloud marketplace in which providers store their offerings. Clients can discover and buy the needed services for third-party cloud applications that they can use for their own mashup. Cloud services will be available through an ad-hoc description language that provides all the relevant aspects for their integration and usage.
- 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.
- There will be a mashup engine, offered as SaaS to clients. It will act as a mediator for atomic cloud services in order to realize user's mashup application.

This vision may have substantial value for IT in this approach by improving the return-on-assets of the existing systems since Mashup enables fast and flexible B2B collaboration (short development cycles, cheap development) whereas existing B2B collaboration solutions focus on long-term business relationships [15]. Short and cheap development cycles make B2B solutions available for small and medium enterprises.

B. Preliminary Definitions

In this subsection we introduce some concepts that are used along the paper. For the sake of clarity the definitions are provided in informal way, however, for the formal counterpart please refer to [10], [12].

The concept of goal is often used in the context of business process. A ***Goal*** is "a desired *change* in the state of the world an actor wants to achieve". Goals represent enterprise strategic interests that motivate the execution of business processes [20]. A *Goal Model* is a requirement engineering conceptual model used to depict the strategic rationale of a business process in the form of a hierarchy of goals. It basically provides a hierarchical decomposition of goals into sub-goals through AND/OR operators. In our approach goal-models are used to specify the business logic of the desired service composition in terms of which outcome the user will receive.

A ***Capability*** is a wrapper for cloud applications and services that introduces a semantic description layer. It allows the developer to specify i) how to invoke the specified functionality (which data must be passed and which data will be returned) and ii) which effect is expected by executing the encapsulated application or service. The capability also has the advantage of being automatically composable in order to address a complex result.

C. A B2B Cloud Application for Fashion Firms

A big manufacturer company who works in fashion (*FashionFirm* [2]) uses a legacy system to manage its information system (IBM AS/400) . They decided to improve their commercial network and designated *Company.com*, a small software house, to handle their B2B processes. The resulting system is built as a set of services running on a cloud stack: a set of scalable backend services (VPN) capable of interacting with the legacy system and a SaaS eCommerce platform (OrderPortal). The provided solution scales up very well for increasing the volume of requests by deploying clustered VPNs on the need.

Given this existing scenario, the Company.com is demanded to extend the FashionFirm's business process by adding new services for customer management. In order to improve cost-effects these new services are conceived as cloud application mashups. This allows to fast prototype the solution by reusing existing third part cloud applications (Cloud Calendar, File Storage and Voicemail).

The mashup application that will be used as a running example in this paper is intended to support customer relationships during the order management process. *ManyFirmsShop* is a retailer of FashionFirm's products. When the ManyFirmsShop agent requests a stock of products (through the OrderPortal), the system merges VPN (used as usual to interact with the legacy system) with external cloud applications existing in the market: a Cloud Storage system to deliver receipts to ManyFirmsShop, Voicemail to communicate the delivery status and, finally, a Cloud Calendar service for annotating the delivery date.

III. The Requirement Awareness

The primary driver of this work is allowing the end-user to build her own mashup over the SaaS infrastructure according to personal requirements. At first analysis the easier choice was to use BPEL, the de facto standard language for orchestrating web-services. However BPEL is a static model, whereas self-configuration allows:

- integrating dynamic user's preferences into the flow of activities;
- introducing new services without revising the whole workflow model;
- service failures may be discovered at run-time.

We used self-configuration in order to configure a Cloud Application Mashup. The most important requisite it demands is the system be requirements-aware. A requirements-aware system should be able to introspect about its requirements in the same way that reflective middleware-based systems permit introspection about their architectural configuration [14]. In a requirement aware system, requirements are at the same time: i) first class abstractions of the modeling language and ii) run-time entities over which the system can assess and reason about.

[2]Names in this scenario are obfuscated for privacy reasons.

In our approach, the GoalSPEC language [13] has been selected for implementing requirement awareness because it uses goals for representing user's requirements and it provides the necessary characteristics for enabling goal injection and reasoning. It represents the first step in the direction of making end-user able to autonomously sketch up her requirements. It is based on structured English and it adopts a core grammar in which a domain vocabulary of terms can be plugged on.

In GoalSPEC a goal is composed of three main compartments: a *triggering condition* (introduced by the 'when' keyword) that states when the goal starts to be interesting for the system, a list of *actors* that are involved in, and a desired *final state* (introduced by the 'address' keyword) that describes what is expected in terms of states of the world. For more details about GoalSPEC see [13].

Listing 1. Set of GoalSPEC goals for the B2B scenario

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

GOAL to_retrieve_user_data: WHEN available(Order)
THE SYSTEM SHALL ADDRESS available(User)

GOAL to_check_order: WHEN available(Order) AND
       available(User)
THE SYSTEM SHALL ADDRESS accepted(Order) OR refused(
       Order)

GOAL to_produce_receipt: WHEN accepted(Order)
THE SYSTEM SHALL ADDRESS available(Receipt)

GOAL to_notify_receipt: WHEN available(User) AND
       available(Receipt)
THE SYSTEM SHALL ADDRESS
MESSAGE receipt SENT TO THE user ROLE

GOAL to_deliver_order: WHEN MESSAGE receipt SENT TO THE
       user ROLE
THE SYSTEM SHALL ADDRESS
MESSAGE delivery_order SENT TO THE storehouse_manager
       ROLE

GOAL to_notify_failure: WHEN available(User) AND
       refused(Order)
THE SYSTEM SHALL ADDRESS
MESSAGE failure SENT TO THE user ROLE
```

An example of goal-model for the FashionFirm scenario is reported in Figure 1, where goals are described in Listing 1. It is worth noting that such a goal model does not provide a logical ordering of the goals to address. It is up to the orchestrator module to deduct the best ordering that applies to the current context.

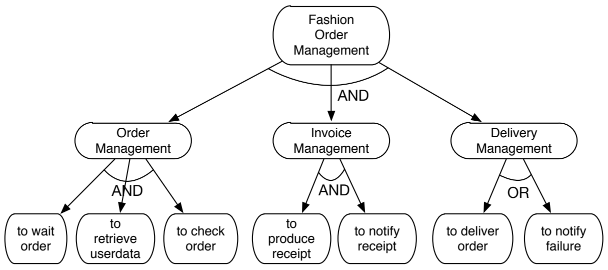


Fig. 1. Goal-model for the FashionFirm scenario.

IV. Self-Configuration

One of the prominent features of autonomic system is self-configuration i.e. the ability to automatically configure its own components thus to ensure the correct global functioning with respect to defined requirements [18].

In a previous work [10] we have introduced the Proactive Means-End Reasoning as the problem of automatically associating system capabilities to user's goals thus to achieve the desired functions. We have called *PMR Ability* the procedure for solving the proactive means-end reasoning problem. By reasoning at the knowledge level the procedure returns 0..n alternative *Configurations* for a given goal-model. It works by composing capabilities in order to match the final state specified by goals thus to solve the user's request.

This section illustrates domain-independent architecture for goal injection and self-configuring of cloud applications.

A. Goal Injection

The proposed architecture handles run-time addition of new requirements [13], [11] moving a step forward traditional systems defined for satisfying a fixed set of hard-coded requirements.

The users may specify new requirements (in the form of goal-models) to inject into the system at run-time thus they become a stimulus for modifying the overall system behavior. It is responsibility of the middleware, via the PMR Ability, to configure itself to the new needs.

On one hand, the system activates a *goal injection monitor* that waits for goals from the user. On the other hand, user-goals are run-time entities, as well as other environment properties. The system acquires goals from the user and maintains knowledge of them thus to be able of reasoning on expected results and finally conditioning its global behavior. Of course, existing goals may be retreated as well.

Goal injection enables *user-requirements to evolve over time* without either user-management or restarting the system. This could be fundamental for some categories of domain in which continuity of service is central (financial, service providing and so on).

In addition it is possible *to increase or enhance the functions of the system* just injecting a new set of requirements and updating the repository with new domain-specific capabilities. Given that connections between goals and capabilities are discovered on demand, the architecture is robust to capability evolution and may be used for different problem domains without any other specific customization.

B. A Three-Layered Architecture for Self-Configuration

The operative hypothesis for self-configuration is the system owns a repository with a redundant set of capabilities, thus being able solve the same problem by exploiting different combinations of capabilities.

The proposed architecture is made of three layers: the goal layer, the capability layer and the business 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 no 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 [10]. 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 concrete 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 ***Business Layer*** manages and interconnects autonomous blocks of computation thus generating a seamless integration for addressing the desired result specified at the first layer. Section V 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.

C. Describing Capabilities for Self-Configuration

Capabilities play a central role in the aforementioned architecture: to compose and execute capabilities the system must know what the service is and that executing the capability would result in a state of the world in which the goal being satisfied. In this approach each capability is a wrapper for a specific service or a cloud application. For instance *Upload on Cloud Storage* is a wrapper for a generic Cloud File Storage HTTP endpoint that creates a new file in a remote path. Each capability is bound to a block of code for authenticating and invoke the specified endpoint.

We adopted a language to specify the ‘self-configuration’ part of a capability with the same the same kind of abstraction used for user’s goals (the knowledge level [8]). The following language, inspired to LARKS [17]) responds to the need to implement reasoning directed towards capabilities [19].

The ‘self-configuration’ description is made of the following compartments:

Name is the unique label used to refer to the capability

Input is the definition of the input Data Objects necessary for the execution. The existence of the specified DataObject is mandatory for enabling the capability execution

Output is the definition of the output Data Objects produced as result.

Params is an optional list of Variables that must be assigned to a value (grounded) in order to enable the execution.

Pre-Condition is a run-time logical condition that must hold in the current state of the world in order to trigger the capability execution.

Post-Condition is a run-time logical condition that must hold after the capability execution for asserting either success or failure.

Evolution is a function that describes at knowledge level the possible impact of the capability for addressing a final state. This function is used in conjunction with grounded params for simulating the use of capabilities for self-configuration purposes [10], [12].

Figure 2 shows an example of capability used by Company.com: *Upload_On_Cloud_Storage* is a wrapper for the Dropbox upload endpoint as discussed before. It receives a document to upload as input, and it requests a token and path as parameters to configure for working. The output is the identification of the remote file.

Name	UPLOAD_ON_CLOUD_STORAGE
Input	Invoice: Document
Output	Remote_id: RxFile
Params	Token: String Dest_path: String
Pre-Condition	*available(Invoice)*
Post-Condition	*uploaded_on_cloud(Invoice)*
Evolution	evo={add(*uploaded_on_cloud(document))}*

Fig. 2. Example of capability for interfacing with a Cloud File Storage.

Figure 3 show a couple of solutions for the FashionFirm scenario. Capabilities are represented in capital letters, followed by assignments for their parameters.

In the first solution (on the left) there is a coincidental direct correspondence between goals and capabilities obtained as the result of self-configuration. In the case the customer does not grant the access to the cloud storage service, the proactive means-end reasoning elaborates a different configuration (on the right) where the goal to_notify_receipt is addressed by uploading the file in a Company.com’s private cloud storage and then by sharing the URL via email.

V. The Run-Time Orchestrator

This section describes the bottom layer of the proposed architecture, in which a set of capabilities is harmonized thus to make the overall behavior consistent with user’s requirements.

The input of this phase is a set of associations $\langle cap, goal \rangle$ generated through the proactive means-end reasoning. The desired output is a workflow model that addresses the root goal of the goal-model.

The orchestrator module works according two principles:

- *Principle 1: Dynamic Association between Capability and Goals*. Each capability selected for addressing a goal is executed according to a specific schema[3].
- *Principle 2: Distributed Control*. All the goals start in parallel and are coordinated by conditions over the state of the world and through exchange of data objects.

[3]More details can be found in [12]

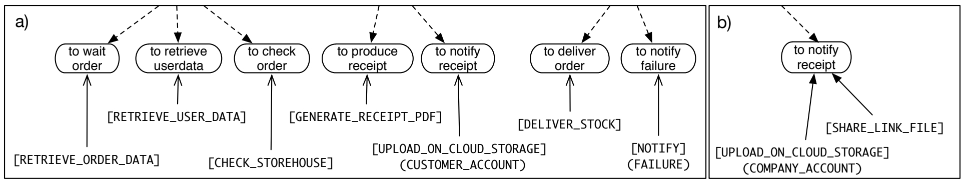


Fig. 3. On the left an example of assignment of capabilities to goals. On the left a detail of an alternative configuration in which a goal is addressed by the composition of a couple of capabilities.

The Principle 1 arises from the fact that each capability is executed because it contributes to address one of the goals of the goal-model. As stated in Section III a goal is described by a triggering-condition (TC) that specifies when it becomes interesting for the system. This is the first condition that must hold for executing the capability. However also the capability has a precondition to verify and some data objects are necessary in order to execute it. At the same way, after the capability is executed, the post-condition reveals whether there is a failure. In addition, by comparing the achieved state of the world with the goal's final state the orchestrator reveals when the goal is addressed.

There is an additional motivation behind the way a capability lifecycle is structure. It already contains all the coordination information for synchronizing the execution of a capability with respect to all the others. Therefore the Principle 2 has been introduced to allow capabilities are deployed in a distributed and scalable system. By executing all the capabilities in parallel generates a distributed BPMN in which each branch represents the instance of a different lifecycles. Different branches interact by two different synchronization approaches: i) passive mode: a branch waits a condition of the state of the world is true, whereas this is generated as final state of another capability of the workflow; ii) active mode: a branch needs a data object as input parameter for the corresponding capability, therefore a Query Interaction Protocol is employed to allows the exchange of data.

It is worth noting the orchestrator may raise signals for self-configuration (capability failure or unexpected state). Please refer to [10] for details about re-configuration.

A. MUSA: a Middleware for User-driven Service Adaptation

The Cloud Application Mashup has been realized through MUSA (Middleware for User-driven Service Adaptation) a multi-agent system for the composition and orchestration of services in a distributed and open environment. MUSA aims at providing run-time modification of the flow of events, dynamic hierarchies of services and integration of user preferences together with a system for run-time monitoring of activities that is also able to deal with unexpected failures and optimization.

MUSA[4], has been implemented by using JASON [3], an agent oriented platform and programming language based events and actions. It implements a distributed version of the proactive means-end reasoning in which the result is a set of capabilities for addressing goals, and also a contract among the agents for working in collaboration. Therefore, service composition is obtained at run-time, as the result of a self-organization phenomenon. More details of the MUSA architecture are reported in [12].

Figure 4 shows some screenshots captured during the execution of the FashionFirm scenario.

[4]Available at https://github.com/icar-aose/MUSA/archive/v0.2.zip (Jason 1.3.8 or higher is required).

VI. Related Work

Some initiatives have been launched in last years for the automated orchestration of Cloud applications. A couple of representative examples are OpenCloudware and FIWARE. OpenCloudware [1] aims at building an open software engineering platform, for the collaborative development of distributed applications to be deployed on multiple Cloud infrastructures (mainly concerned with IaaS and PaaS). FIWARE is an open architecture and a reference implementation of a novel service infrastructure [6] 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 in multiple sectors". It offers an application mashup platform aimed at integrating heterogeneous data, application logic, and UI components (widgets) sourced from the Web.

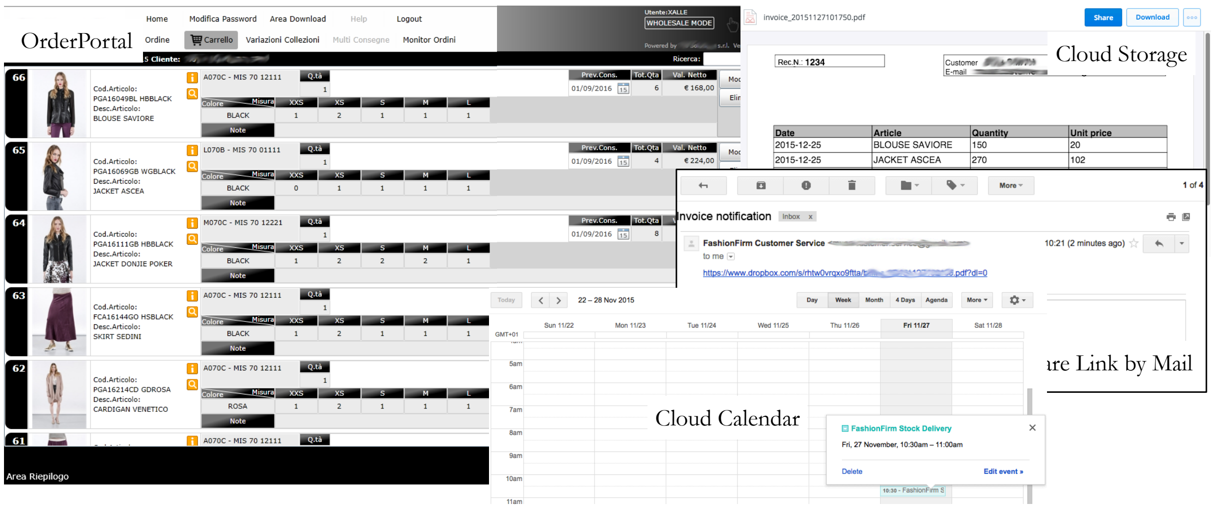
The presented approach can be classified among the alternative approaches to classic workflow models for describing service compositions, for instance semantic type matching approach for creating or updating a workflow [7] or planning based for composing a choreography [4].

Blanchet et al. [2] view service orchestration as a conversation among intelligent agents, each one responsible for delivering the services of a participating organization. An agent also recognizes mismatches between its own workflow model and the models of other agents.

OSIRIS [16] is an Open Service Infrastructure for Reliable and Integrated process Support that consists of a peer-to-peer decentralized service execution engine and organizes services into a self-organizing ring topology.

VII. Conclusions

This paper has described the practical experience of Cloud Application Mashup, made in the context of a research project. We adopted MUSA as the middleware for service composition and adaptation. This allowed defining the business logic of



Cliente: [redacted]

								Prev.Cons.	Tot.Qta	Val. Netto
66	Cod.Articolo: PGA16049BL HBBLACK Desc.Articolo: BLOUSE SAVIORE	A070C - MIS 70 12111	Q.tà 1					01/09/2016	6	€ 168,00
		Colore / Misura	XXS	XS	S	M	L			
		BLACK	1	2	1	1	1			
65	Cod.Articolo: PGA16069GB WGBLACK Desc.Articolo: JACKET ASCEA	L070B - MIS 70 01111	Q.tà 1					01/09/2016	4	€ 224,00
		Colore / Misura	XXS	XS	S	M	L			
		BLACK	0	1	1	1	1			
64	Cod.Articolo: PGA16111GB HBBLACK Desc.Articolo: JACKET DONJIE POKER	M070C - MIS 70 12221	Q.tà 1					01/09/2016	8	
		Colore / Misura	XXS	XS	S	M	L			
		BLACK	1	2	2	2	1			
63	Cod.Articolo: FCA16144GO HSBLACK Desc.Articolo: SKIRT SEDINI	A070C - MIS 70 12111	Q.tà 1							
		Colore / Misura	XXS	XS	S	M	L			
		BLACK	1	2	1	1	1			
62	Cod.Articolo: PGA16214CD GDROSA Desc.Articolo: CARDIGAN VENETICO	A070C - MIS 70 12111	Q.tà 1							
		Colore / Misura	XXS	XS	S	M	L			
		ROSA	1	2	1	1	1			
61		A070C - MIS 70 12111	Q.tà							

invoice_20151127101750.pdf

Rec.N.: **1234**

Customer [redacted]
E-mail [redacted]

Date	Article	Quantity	Unit price
2015-12-25	BLOUSE SAVIORE	150	20
2015-12-25	JACKET ASCEA	270	102

Invoice notification

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

FashionFirm Stock Delivery
Fri, 27 November, 10:30am – 11:00am
10:30 - FashionFirm S

Fig. 4. Screenshots of the mashup application: OrderPortal is the web application for warehouse orders, a Cloud Storage service is adopted where the invoice is uploaded, a mail service allows to send the link to the remote invoice, and a Cloud Calendar service is used for annotating the stock delivery date.

the mashup in the form of goal model plus a set of Goal-SPEC specifications. On the other side the involved Cloud applications have been decomposed in a set of components to wrap into capabilities and to describe through a template based on descriptive logic. The middleware offers a three layer-architecture for monitoring goal injections, self-configuring ad-hoc solutions and finally to orchestrate Cloud components. The approach has been employed in the context of a B2B process for customer management of a big company working in fashion.

References

[1] 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.

[2] 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.

[3] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2007.

[4] 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.

[5] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: An organizational view of multi-agent systems. In *Agent-Oriented Software Engineering IV*, pages 214–230. Springer, 2004.

[6] A. Glikson. Fi-ware: Core platform for future internet applications. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[7] M. Laukkanen and H. Helin. Composing workflows of semantic web services. In *Extending Web Services Technologies*, pages 209–228. Springer, 2004.

[8] A. Newell. The knowledge level. *Artificial intelligence*, 18(1):87–127, 1982.

[9] M. P. Papazoglou and W.-J. van den Heuvel. Blueprinting the cloud. *IEEE Internet Computing*, 6:74–79, 2011.

[10] 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.

[11] 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.

[12] 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.

[13] L. Sabatucci, P. Ribino, C. Lodato, S. Lopes, and M. Cossentino. Goal-spec: A goal specification language supporting adaptivity and evolution. In *Engineering Multi-Agent Systems*, pages 235–254. Springer, 2013.

[14] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 95–103. IEEE, 2010.

[15] 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.

[16] 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.

[17] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous agents and multi-agent systems*, 5(2):173–203, 2002.

[18] 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.

[19] M. J. Wooldridge. *Reasoning about rational agents*. MIT press, 2000.

[20] 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.

[21] 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.