Highly Customizable Service Composition and Orchestration

Luca Sabatucci, Carmelo Lodato, Salvatore Lopes, Massimo Cossentino

ICAR-CNR, Palermo, Italy
{sabatucci,c.lodato,s.lopes,cossentino}@pa.icar.cnr.it

Abstract. One of the current challenges of Service Oriented Engineering is to provide instruments for dealing with dynamic and unpredictable user requirements and environment. Traditional approaches based on workflow for orchestrating services provide little support for configuring at run-time the flow of activities.
This paper presents a general approach for composing and orchestrating services in a self-organization fashion. User requirements are made explicit in the system by a goal specification language. These can be injected into the running orchestration system that is able to autonomously and contextually reason on them. Therefore, the system dynamically organizes its structure for addressing the result. A prototype of the system has been implemented in JASON, a language for programming multi agent systems. Some aggregate statistics of execution are reported and discussed.

Keywords: Service Orchestration, Self-Organization, Holonic System

1 Introduction

In the last years there has been an increasing interest on composing and orchestrating heterogeneous services from many parties. To date, BPEL is the de facto standard for implementing the orchestration of services. Even if greatly supported by industry and research, the classic workflow approach is not easy to extend for supporting some advanced features: 1) integrating user's preference into the flow of activities complicates much the model; 2) there is no a simple way to change the flow of activities as consequence of a change of the execution context; 3) introducing new services requires to revise the whole workflow model; 4) service failures may be included in the design but any unexpected situations make the process fails. It is a fact that researchers are also investigating on dynamic workflows and on techniques for generating a highly configurable system behavior, potentially adaptable to unexpected events [5, 9].

The assumptions of this work are that: i) services are delivered over the internet by service providers; as usual, these are accessible through standards protocols (i.e. WSDL and SOAP); (ii) the orchestration system is a distributed and decentralized software, made of a number of autonomous agents, each able to perceive the environment and act as broker for web-services (of which it knows

address, end points and business logic); and (iii) holons are temporary assembly of agents generated ad-hoc for aggregating services.

In this paper we propose a middleware for conciliating goal-orientation [24] with holonic systems [11] with the aim of creating a highly customizable orchestration of web-services. Goal orientation is used for decoupling the specification of what the system has to do from how it will be done. The request for a service 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. Once it has been specified, the goal may be injected into the system at run-time, thus becoming a stimulus for the holons of the system that try to self-organize in an ad-hoc architecture for fulfilling the request.

On the other side, holons provide an elegant and scalable method to design and develop a distributed software system with a natural inclination for knowledge sharing, coordination of activities and robustness. The goal specification language is enough flexible to create complex requests that any available service may satisfy alone. However the developer has to code only simple interactions with basic services: service compositions must not be programmed. It is responsibility of the system that of aggregating basic services thus to obtain composed ones.

The paper is organized as follows: Section 2 provides an overview of the proposed middleware. Section 3 presents some preliminary definition by introducing the holonic approach. Section 4 provides details on how services are aggregated and orchestrated for addressing a set of user-goals. Section 5 illustrates the state of the art in service composition and orchestration, whereas Section 6 presents a critical analysis of the approach. Finally, Section 7 briefly summarizes the impact of the work.

2 Overview: the MUSA Approach

This paper presents MUSA (Middleware for User-driven Service Adaptation) [1], a holonic multi-agent system for the composition and the orchestration of services in a distributed and open environment. The middleware aims at providing run-time modification of the flow of events, dynamic hierarchies of services and integration of user preferences together with a self-adaptive system for execution activities that is also able to monitor unexpected failures and to reschedule in order to optimize the flow.

The main feature of the system is to break the static constraints of a classic workflow model by decoupling the two dimensions: 'what to address' and 'how to address it' [21]. The core element of the approach is the use of *Goals* for explicitly representing user-preferences into the system (what to address). The injection of goals trigger for the re-organization of the agents of the system in hierarchical groups called holons. These self-adaptive structures allow for dealing with dynamic composition and orchestration of services.

[1] Website: http://aose.pa.icar.cnr.it/MUSA/.

The Capability is the key for addressing the injected goals coming from the decomposing of standard workflow into a set of atomic and self-contained parts to connect in many different ways. An illustrative example is reported in Figure 1 that shows a portion of the BPEL for a travel reservation workflow. Tasks related to the flight reservation (search, booking, canceling) as well as those related to hotel reservation are highlighted.

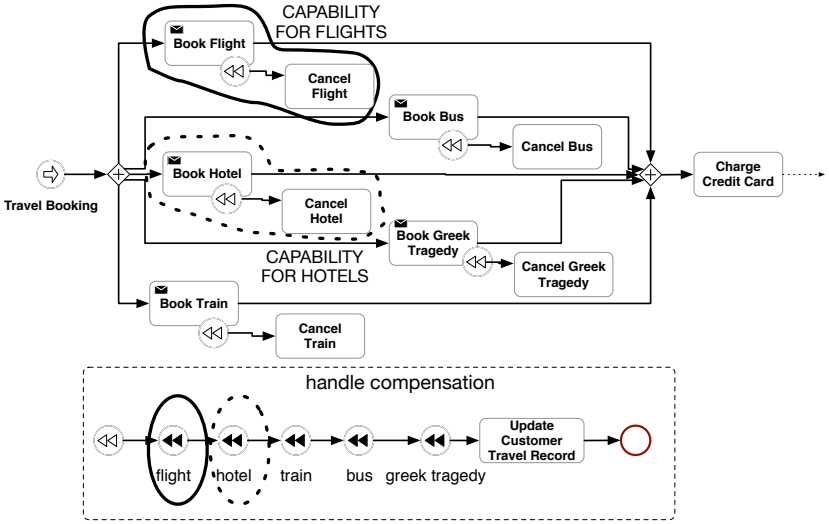

Fig. 1. An example of workflow for booking a set of related services for a travel The capability approach consists in identifying sub-parts to decompose for obtaining self-contained pieces of behavior of the system.

As a running example we refer to the *smart travel system*, able to act as touristic tour operator in a geographical area. A scenario will help in clarifying variability and flexibility required in a service composition context.

Scenario: *Herbert from Munich wants to organize a vacation in Sicily for a week with his family. In seven days, Herbert desires: 1) to visit the city of Palermo; 2) to visit Agrigento and 3) to visit Syracuse and to attend the Greek tragedy performance. The smart_travel service suggests to flight to Palermo, stay there 3 days, than to visit Agrigento for 2 days, then to move to Syracuse (for the Greek tragedy) and finally to depart from the Catania airport. Herbert confirms the travel plan, and the smart_travel reserves flights and hotels and buys tickets for the trip.*

The MUSA middleware offers a suitable infrastructure for implementing the smart travel system. Figure 2 provides an overview of the customization of MUSA for the specific domain. The user will interact with the smart travel through a web page for specifying her preferences (dates, places to visit, events

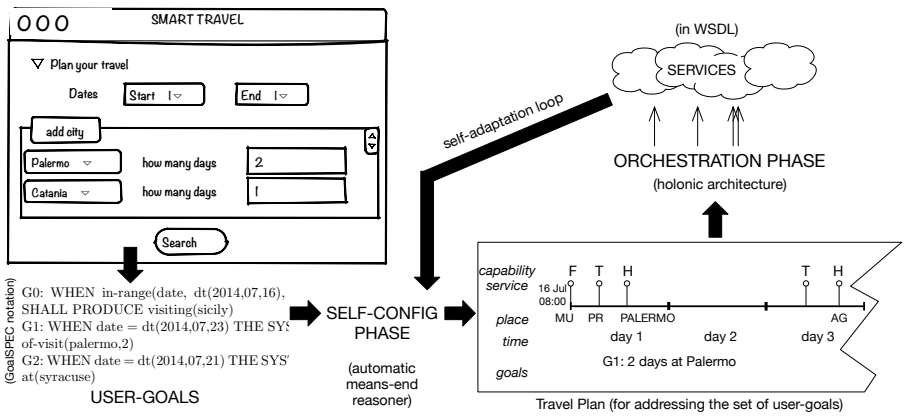

Fig. 2. An example of the MUSA approach to the case study of the smart travel.

to assist, and other interests). The web page converts user-data into a set of goals in an ad-hoc language called GoalSPEC [22]. Goals are injected into the MUSA running system that interprets them and uses them as directives for a planning phase. This algorithm, called automatic means-end reasoning [20], is responsible of discovering 0..n solutions for addressing the user-goals. A solution represents a collection of web-services and a semantic layer working as instructions for orchestrating them. Selecting one of these solutions triggers the organization of holonic architecture for orchestrating services and monitoring the state of interest. The system is able of identifying is some of the selected services fails. If this happens the self-adaptation mechanism calls again the planning algorithm for re-organize the architecture.

3 Preliminary Definitions

Holons provide an elegant and scalable method to guarantee knowledge sharing, distributed coordination and robustness.

3.1 A Brief Introduction to Holons

Holon is a Greek word for indicating something that is simultaneously a whole and a part [16]:

> A holon is a system (or phenomenon) that is an evolving self-organizing structure, consisting of other holons [15]. A holon has its own individuality, but at the same time, it is embedded in larger wholes (principle of duality or *Janus effect*).

Many concrete things in nature are organized as a holarchy (the recursive structure generated by holons and sub-holons). An example of concrete holon is an *organ* that is a part of an organism, but a whole with regard to the cells of which it is comprised. The human mind uses holarchies for organizing abstract concepts too. An example is a *word* that is part of a sentence, but a whole with regard to the letters that compose it.

A holon has not necessarily the same properties of its parts, as well as if a bird can fly, its cells can not. Holon is therefore a general term for indicating a concrete or abstract entity that has its own individuality, but at the same time, it is embedded in larger wholes.

In computer science the holon has been recently employed [8] to represent software as dynamic groups of autonomous software entities. A *holonic multi agent system* is a software system made of autonomous agents who organize themselves in holons.

Definition 1 (Holon). *A holon is a triple* $h = \langle Head, Body, Commitments \rangle$ *where* $Body = \{h_1, h_2, \ldots h_n\}$ *and* $\forall i, h_i$ *is a holon and sub-holon of* h*. The* $Head \subseteq Body$ *is the entity in charge of representing and coordinating the actions inside* h *and, finally,* $Commitments$ *are relationships inside the holon that aggregate the parts* h_i *towards common objectives. The base for the recursion is the agent which represents a sort of atomic holon of the system (a holon that can not be further decomposed in sub-holons): if* a *is an agent of the system,* $h_a = (\{a\}, \{a\}, \emptyset)$ *is the corresponding atomic holon.*

In our approach we map 1) the distributed nature of services to agents of the system and 2) the composability of services to holons. Therefore each atomic holon is in charge of dealing with a single service, whereas higher level holons can handle compositions of services. Holons are not defined according design-time schema. They are able to generate super-holons for dealing with complex problems on demand.

3.2 Knowledge, Goals and Capabilities

Each holon of the system maintains a knowledge of the context of execution. We adopt a frame-based first order logic model of knowledge: (Bel h φ) specifies that the holon h believes φ is true, where φ is a first order fact.

In a holon, knowledge is maintained by the head adopting a structure called *State of World* and it is shared on request to sub-holons.

Definition 2 (State of the World). *A subjective 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-contradictory and non-negated) facts* $(s_1, s_2 \ldots s_n)$ *that can be asserted to describe a given domain.*

W^t describes a closed-world in which everything that is not explicitly declared is assumed to be false.

The peculiarity of this holonic system is that the commitment relationship (e.g. the glue that puts together a holon) is a run-time property called User-Goal. A goal is a specification of the expected behavior of the whole system.

Definition 3 (Goal). *A goal is a pair: $\langle tc, fs \rangle$ where tc and fs are logical formula indicating, respectively the trigger condition (i.e. when the goal may be actively pursued) and the final state (i.e. when the goal may be considered addressed). The truth table of these conditions is evaluated (by unification) through facts of the current state of the world.*

Goals can be used to specify the business logic of the desired service composition in terms of which outcome the user will receive. In MUSA the goal may be injected at run-time by using GoalSPEC see [22], a goal specification language that integrates a subset of natural language and first-order predicates. Examples of goals for the smart travel agency:

- G0: WHEN date IS BETWEEN dt(2014,07,16) AND dt(2014,07,23) THE SYSTEM SHALL PRODUCE visiting(sicily)
- G1: WHEN date IS dt(2014,07,23) THE SYSTEM SHALL PRODUCE day-of-visit(palermo,2)
- G2: WHEN date IS dt(2014,07,21) THE SYSTEM SHALL PRODUCE being-at(syracuse)
- G3: WHEN date IS dt(2014,07,21) THE SYSTEM SHALL PRODUCE enjoyed(greek-tragedy)

The WHEN reserved word is used for specifying an external event (in the example it introduces the goal triggering condition clause), whereas the SHALL PRODUCE reserved words specifies the final state of the world that is desired by the user. More details about this specification language are given in [22].

In the following we focus on illustrating that when the user defines and injects a set of goals, then the system generates a holonic architecture as response.

To this aim we need to introduce the concept of ***Capability*** as an atomic and self-contained action the agent knows to have and how to use it. A capability is a run-time property that holons may intentionally use to interact with a web-service.

In MUSA, a capability is concretely realized by two macro-components: (i) the abstract description that is a sort of 'manual' about the usage of the service, in a self-aware fashion, and (ii) the concrete implementation is the machine-code for invoking a specific web-service.

For what concerns the abstract description, the most significant properties are pre and post (conditions) to be uses in the orchestration phase, and the params and evolution to be used in the self-configuration phase for composing services thus to address complex problems that single services cannot afford.

The "pre-condition" must be true, when tested in the current state of the world ($pre(W^t) = true$), in order to execute the capability. The "post-condition" must be true, after the execution of the capability ($post(W^{t+1}) = true$), for assessing its correct execution. On the other side the "params" describes points of variability of the service for allowing self-configuration (see 4.1), whereas the "evolution" describes the endogenous effect of a capability in terms of changes of state of world ($W^t \rightarrow W^{t+1}$). Figure 3 shows an example of abstract description of the flight booking capability.

Name	FLIGHT_BOOKING	Knowledge about:
Input	DPTPLACE : AIRPORT, DPTDATE: DATE, ARRPLACE : AIRPORT, PASSNUM : INTEGER	*data consumed* in order to properly work, and *data produced* as result of the work
Output	DPTSCHEDULE: DATE, ARRSCHEDULE: DATE	
Constraints	$DptPlace \neq ArrPlace$ $DptSchedule > DptDate$ $ArrSchedule > DptSchedule$ $PassNumber > 1$	
Params	FLIGHID: STRING	how to customize the capability
Pre-Condition	$seat\_available(FlighId, DptSchedule, PassNum)$	*when* the capability may be executed and *what* to expect as result
Post-Condition	$flight\_booked(FlighId, DptSchedule, PassNum)$	
Evolution	$evo = \{remove(being\_at(DptPlace)), add(being\_at(ArrPlace))\}$	how to change the state of the world

Fig. 3. An example of *abstract description* of a capability for dealing with flight booking web-service.

On the other side, the concrete implementation encapsulates the code for interacting with the real service by using SOAP and WSDL through the HTTP/ HTTPS protocol. The implementation of a capability is different for internal capability and webservice-capability. In particular the implementation of a web-services includes three parts: the *customize helper*, the *dispose service* and the *compensate service*. Figure 4 shows the three protocols used in the flight booking activity. They are detailed in the next section.

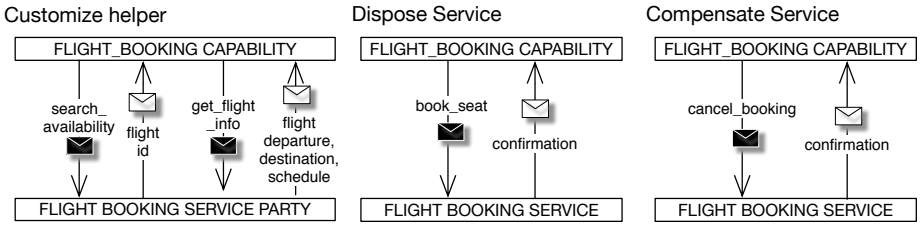

Fig. 4. The concrete implementation of a capability for dealing with flight booking activity.

4 Adaptive Composition and Orchestration of Services

This section assumes that MUSA has been instrumented with a set of capabilities for working in a specific problem domain. For instance, in the case of the smart travel, the capabilities may reserve flights, book hotels, buy ticket for a local event and so on.

When a user introduces a set of goals into MUSA for requesting the personalized execution of services, then the system will firstly discover i) how to compose its capabilities for addressing all the goals, and subsequently ii) it orchestrate the agents of the system thus to allow them properly using their capabilities.

4.1 Self-Configuration Phase

The Self-Configuration phase starts soon after that a set of goals is injected into the system. It aims at discovering a set of capabilities (among the available ones) which composition potentially leads to the achievement of all the goals. In particular, the problem is: given the current state of the world, a set of Goals and a set of Capabilities to produce a plan for addressing the goals. The procedure is called Proactive Means-End Reasoning [20].

A complete description of the algorithm is out of the scope of this paper. Conversely, an exemplar scenario of execution is reported for providing an intuition of the approach used for composing the capabilities. Supposing the current state of the world is:
$W_I = [at(palermo), it\_is(morning)]$:

1. simulate the use of the capability Visit_City_HalfDay:
   $[at(Place), visited(Place, X)] \rightarrow [at(Place), visited(Place, X + 0.5)]$
   for producing the new world:
   $W_1 = [at(palermo), it\_is(afternoon), visited(palermo, 0.5)]$
2. compare the previous result with the state of the world due to the use of a different capability: Book_Train:
   $[at(palermo), it\_is(morning)] \rightarrow [at(catania), it\_is(afternoon)]$
   that generates:
   $W_2 = [at(catania), it\_is(afternoon), visited(palermo, 0.5)]$
3. given that first path is more promising (with respect of the goal to visit Palermo for two days) then discard the second and proceeds from W_1
4. concatenate Visit_City_HalfDay with Reserve_Hotel:
   $[at(Place), it\_is(afternoon)] \rightarrow [at(Place), it\_is(morning)]$
   for obtaining
   $W_3 = [at(palermo), it\_is(morning), visited(palermo, 0.5)]$.
5. ...

In this context, the *customize helper* protocol of the involved capabilities is employed for exploring the range of possible impact that each individual capability may have towards the evolution of the state of the world. Indeed a capability

may include, in its description, some parameters to configure for obtaining different results. For instance, the flight_booking capability may be customized by assigning a value to the flight_id param: FLIGHT_BOOKING[$flight\_id \leftarrow$ "AZ243"]. A different configuration for this parameter leads to different effects (departure/destination places, timetable, flight company and costs may be sensibly different).

During the Self-Configuration phase, the customize helper considers many parameters for a capability, in order to configure it for the specific context. This generally requires interacting with real web-services for querying the range of possible values for the parameters. For instance, the customize helper for the flight_booking searches for available flights that may be useful in the context of user's travel (existing flight from a destination to a target city, in a given date with available seats). It returns a list of possible Flight IDs that satisfy the input conditions: each item corresponds to a possible variant of the capability.

For a detailed description of the procedure, please refer to [20].

4.2 Service Orchestration Phase

The selection of a solution (for addressing a set of goals) triggers for a holon formation and it promotes the corresponding holon to become operative and to orchestrate all the embedded services for producing the compounded result. For instance, let us suppose the holon represented in Figure 5. It is made of four sub-holons: (H_{PA} able to address the goal G1, H_{SY} able to address the goals G2/G3 and H_{AG}, H_{CT} created for completing the travel).

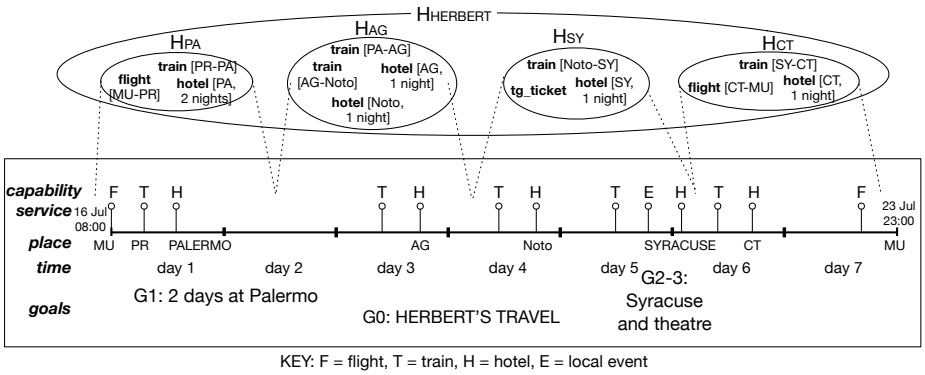

Fig. 5. In the topside: an instance of Holon ($H_{HERBERT}$) formed for addressing Herbert's travel specifications. It is composed of four sub-holons (H_{PA}, H_{AG}, H_{SY} and H_{CT}). Finally, atomic holons for simplicity are shown as the capability they offer (in bold text), followed by their params. Below: the corresponding travel plan showing the schedule and highlighting where Herbert's goals are satisfied.

When a holon becomes active, then (recursively) all its sub-holons become active. From the point of view of the atomic holon, this corresponds to execute the *dispose* protocols of service capabilities, as soon as the capability pre-conditions hold. It is worth noting that since agents are autonomous and distributed, all the dispose protocols will be executed by parallel threads. Figure 6 represents the corresponding flow of activities resulting from the holon orchestration of capabilities. In the case of the smart travel, the holon will book all necessary flights, hotels, ticket for the travel plan. In particular, the dispose protocol for the flight_booking service actually book the specified flight and produces a ticket for the user.

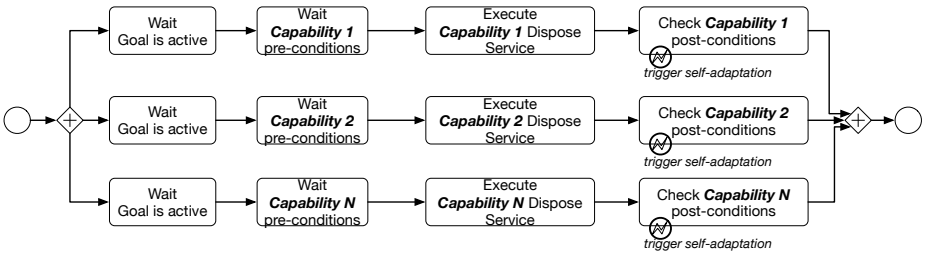

Fig. 6. The corresponding flow of activities resulting from the composition of three capabilities for addressing a goal.

In addition, when possible, holons will activate their monitoring capabilities for checking the real service execution. This detail is out of the scope of this paper. Just to provide an example, by interacting with Herbert's mobile applications the monitor agent may know his position, and interact with him for warning about delays or allowing to change details of the travel.

4.3 Self-Adaptation

The main purpose of monitoring the services is to look for failures or new goals that may affect the running holon. When something unexpected happens at run-time, it could be the case that some services that have been disposed are no more useful in the new plan. The proper way to proceed is to use the *compensate service* protocol in order to terminate the contract with a service. For instance, the compensate service for the flight_booking tries to cancel the user booking for a specified flight.

The holon in charge of addressing a goal is continuously updated about the state of the services and it is able to discover when something is going wrong by comparing perceptions with expected states of the world. When a service fails, or when a goal cannot be addressed then the head role of the corresponding sub-holon raises an event of failure (see Figure 6), which is the cause of an adaptation.

Let us consider the following variability scenario, concerning the smart travel system described in Section 3: *Herbert and his family are enjoying their vacation in Sicily. They are visiting Palermo and communicate to the smart_travel service their new desire to stay one day more in the city.*

In this example the user who modified her goals triggers the adaptation (by informing the system to change the travel plan). The adaptation is treated by the system as a reorganization of the holonic architecture. The reorganization produces a temporary disassembly of the holon and the re-execution of the Self-Configuration phase but considering the new situation (current state of the world, failures, service availability or new user goals) for generating a new solution.

Continuing the example, the smart travel system reacts to the new goal considering the current state of the trip, and that theater tickets (of the sixth day) are not reimbursable. Therefore the system proposes: 1) to stay 1 day more in Palermo, 2) to stay one day only at Agrigento, then 3) to move to Syracuse and continue the vacation without further variations. If Herbert confirms the new travel plan then the smart_travel will change train booking and hotel reservations.

Before starting the new solution, each holon coordinates with its head for deciding if executing the *compensate* protocols of capabilities associated to the services that are no more useful in the new solution. After that, the orchestration phase starts again.

5 Related Work

In last decade, researchers have been looking for alternative approaches to classic workflow models for describing service compositions. For instance, Laukkanen and Helin [17] illustrate a semantic type matching approach for creating or updating a workflow. Traverso et al. [6] show that an instance of the service choreography problem can be viewed as a STRIPS-style planning problem in which state descriptions are ambiguous and operator definitions are incomplete. Whereas approaches based on planning are NP-Complete, Doshi et al. [10] propose using Markov decision processes and Bayesian model learning to model workflow composition with a polynomial complexity. Buhler and Vidal [4] present an introductory work on adaptive workflow composition based on a multi-agent perspective. They suggest the utilization of standard workflow languages for multi-agent coordination.

SAPERE [25] (Self-Aware Pervasive Service Ecosystems), is a general framework for self-organizing distributed service ecosystems. Components of the system can inject Live Semantic Annotations that propagate to other components, while EcoLaws define how they interact in the ecosystem.

A-3 [1] is self-organizing distributed middleware aiming at dealing with high-volume and highly volatile distributed systems. It focuses on the coordination needs of complex systems, yet it also provides designers with a clear view of

where they can include control loops, and how they can coordinate them for global management.

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 own workflow model and the models of other agents.

Gomaa and Hashimoto, in the context of the SASSY research project [12], look into software adaptation patterns for Service-Oriented Applications. The goal is to dynamically adapt distributed transactions at run-time, separating the concerns of individual components of the architecture from concerns of dynamic adaptation, using a connector adaptation state-machine.

OSIRIS [23] 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.

Grassi et al. [13] propose a QoS-aware decentralized service assembly based on the 3 architectural layers. They concentrate their contributions on the middle layer (change management). 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. Plus, non-functional requirements and only the currently available services should be considered. Even further, all of this should be done using decentralized self-assembly (no external control, dynamic operation, no central control).

Hahn and Fischer, in [14] illustrate how a choreography model can easily be conceptually implemented through holons. Their approach is a design-to-code technique based on model-driven transformations which result is a holonic multi-agent system.

6 Discussion and Evaluation

The system[2] has been implemented by using JASON [3], an agent oriented platform based on AgentSpeak [18]. AgentSpeak is a programming language based on events and actions. The state of an agent together with its environment and eventual other agents represent its belief base. Desires are states that the agent wants to attain based on its perceptions and beliefs. When an agent adopts a plan it transforms a desire to an intention to be pursued. In JASON, the agent's knowledge is expressed by a symbolic representation by using beliefs, which are simple predicates that state what the agent thinks to be true.

Completeness and Complexity of the Approach. The current algorithm used for implementing the means-end reasoning is a variation of the depth-first search strategy. Indeed, at the worst case, it takes an exponential time to visit all the possible states.

We accepted this complexity because we observed that in a real situation rarely, given a state of the world, there are too much competing services that may

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

solve a goal. Therefore we assumed to explore only a limited space of solutions 1) by filtering in advance capabilities that are impossible to compose (according to a preliminary evaluation of preconditions) and 2) by employing domain-specific utility functions to measure, step by step, the quality of the partial solutions.

We conducted an experiment in which we requested the smart travel system to organize available services for a 7 days vacation in Sicily. The algorithm has been executed 50 times with five different sets of user-goals in order to evaluate the number of steps necessary for discovering (at most) 5 travel plans; in all the cases the execution returned 5 solutions by exploring, in average, 73 states of world.

We also compared the average branch factor with respect to a simple depth-first algorithm in which the number of capabilities (6 in the experiment) is equal to the branch factor. The resulting average branching factor is 2.65, therefore at each step more than 70% of the capabilities are discarded, thus reducing the space to explore.

Table 1. Aggregated statistics for the execution of the self-configuration phase for the smart travel. The algorithm has been executed 50 times with a range of 5 different configurations (changing the number and the type of user-goals from a minimum of 2 to a maximum of 5) but with a constant number of capabilities, set to 6.

First Solution	44,00 states
Total States of World	73,25 states
Max Depth	17,25 capabilities
Average Branch Factor	2,65 states
Max Number of Partial Solutions	76,75 part. solutions

Ontology Commitment and System Evolution. Another point of discussion concerns the degree of decoupling between Capabilities and Goals. These are specified in two independent languages, and injected into the system at runtime. In addition they can evolve during the time, thus making the whole system able to encapsulate new functionality on the fly. However the use of ontology is required for enabling a semantic compatibility between Capabilities and Goals. An ontology is the specification of a conceptualization made for the purpose of enabling knowledge sharing and reuse.

The Problem Ontology Diagram (POD) [8] may be used to provide a denotation to significant states of the world thus giving a precise semantics to goals and capabilities. A POD is a conceptual model [19] to create an ontological commitment between developers of capabilities and users who inject goals, i.e. the agreement to use a thesaurus of words in a way that is consistent with respect to the theory specified by ontology.

This artifact aims at producing a set of concepts, predicates and actions and at creating a semantic network in which these elements are related to one another. The representation is mainly human-oriented but it is particularly suitable

for developing cognitive system that are able of storing, manipulating, reasoning on, and transferring knowledge data directly in first-order predicates [19].

Grounding goals and capability abstract description on the same ontology is fundamental to allow the system to adopt a proactive means-end reasoning to compose plans. By committing to the same ontology, capabilities and goals can be implemented and delivered by different development teams and at the same time enabling a semantic compatibility between them.

More details on the POD are in [8], whereas the link between goals and ontology is detailed in [19]. Finally we also provide GIMT (Goal Identification and Modeling Tool) a tool for supporting ontology building and goal modeling [7].

7 Conclusions

Holonic multi-agent systems provide a flexible and reconfigurable architecture to accommodate environment changes and user customization. This paper has presented a dynamic (re-) organization of the system by an autonomous and proactive collaboration of autonomous agents. The novelty of the proposed approach, with respect to the state of the art, is to encapsulate the desired service composition in run-time goal-models. Goals are injected into the system thus allowing holarchy to spontaneously emerge for orchestrating services that will address them.

8 Acknowledgment

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

References

1. L. Baresi and S. Guinea. A3: self-adaptation capabilities through groups and coordination. In *Proceedings of the 4th India Software Engineering Conference*, pages 11–20. ACM, 2011.
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. P. A. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information technology and management*, 6(1):61–87, 2005.
5. P. A. Buhler, J. M. Vidal, and H. Verhagen. Adaptive workflow= web services+ agents. In *ICWS*, volume 3, pages 131–137, 2003.
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. 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.
8. M. Cossentino, N. Gaud, V. Hilaire, S. Galland, and A. Koukam. Aspecs: an agent-oriented software process for engineering complex systems. *Autonomous Agents and Multi-Agent Systems*, 20(2):260–304, 2010.
9. M. Dixit, A. Casimiro, P. Lollini, A. Bondavalli, and P. Verissimo. Adaptare: Supporting automatic and dependable adaptation in dynamic environments. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(2):18, 2012.
10. P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition using markov decision processes. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 576–582. IEEE, 2004.
11. 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.
12. 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.
13. 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.
14. 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.
15. J. J. Kay and M. Boyle. *Self-organizing, holarchic, open systems (SOHOs)*. Columbia University Press: New York, NY, USA, 2008.
16. A. Koestler. The ghost in the machine. 1967. *London: Hutchinson*, 1967.
17. M. Laukkanen and H. Helin. Composing workflows of semantic web services. In *Extending Web Services Technologies*, pages 209–228. Springer, 2004.
18. A. S. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996.
19. P. Ribino, M. Cossentino, C. Lodato, S. Lopes, L. Sabatucci, and V. Seidita. Ontology and goal model in designing bdi multi-agent systems. *WOA@ AI* IA*, 1099:66–72, 2013.
20. 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*, May 18-19 2015, Florence, Italy.
21. 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.
22. 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.
23. 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.
24. 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.
25. F. Zambonelli, G. Castelli, M. Mamei, and A. Rosi. Programming self-organizing pervasive applications with sapere. In *Intelligent Distributed Computing VII*, pages 93–102. Springer, 2014.