MUSA: a Middleware for User-driven Service Adaptation

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

***Abstract*—One of the current challenges of Service Oriented Engineering is to provide instruments for dealing with dynamic and unpredictable environments and changing user requirements. Traditional approaches based on static workflows provide little support for adapting at run-time the flow of activities.**

MUSA (Middleware for User-driven Service Adaptation) is a holonic multi-agent system for the self-adaptive composition and orchestration of services in a distributed environment.

I. Introduction

In the last decade web-services have gained industry-wide acceptance as the universal standard for enterprise application integration [1]. Their strengths is to be easily combined as building blocks of a large distributed and scalable software application [2]. On the other side, fostering user participation in business process is an enormous opportunity, where the value is direct and proportional to the capability to customize service parameters according to users' needs [3].

To date, BPEL is one of the most used standards for implementing the orchestration of services. Even if workflow-based languages are greatly supported by industry and research, their approach reveals being static and not easy to extend for supporting some advanced features as, for example, run-time modification of the flow of events, dynamic hierarchies of services, integration of user preferences and, moreover, it is not easy to provide a system for run-time execution, rescheduling and monitoring of activities that is also able to deal with unexpected failures and optimization.

Recently, the research community on services has been very active in defining techniques, methods and middleware for supporting dynamic execution model for workflows.

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.

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

MUSA 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.

Self-adaptation is based on the intuition to break the static constraints of a classic workflow model by decoupling the 'what' (the outcome the workflow requires to be addressed) from the 'how' (the way this result can be obtained) [4].

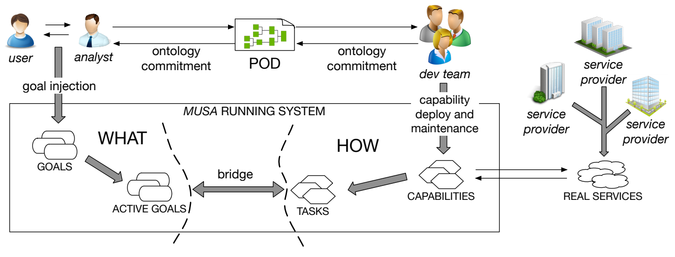


Figure 1. The MUSA vision.

Figure 1 illustrates the underlying idea. Services are provided as usual by world-wide companies, according to their own business processes. MUSA provides a platform in which 1) virtual enterprises can deploy some capabilities that wrap real services, completing them with a semantic layer for their smart use; 2) analysts and/or users can inject their goals for requesting a specific outcome. Under the hypothesis that both goals and capabilities refer to the same semantic layer (described as an ontology), then agents of the system are able to conduct a proactive means-end reasoning for composing available capabilities into task for addressing the user request.

Conceptually, this has required the following ingredients: *goal-orientation* for making user-requirements explicit in the system thus breaking the strict coupling among activities of the workflow; *holonic system*, for implementing a dynamic and re-configurable architecture of autonomous and proactive agents.; *self-adaptation*, for generating smart and dynamic plans as response to user-requests and to unpredictable events of the environment.

The whole system has been implemented by using JASON [5] and CArtAgO [6], an agent facility based on the AgentSpeak language [7] and the BDI theory [8].

The running example used along the whole paper concerns the domain of *Travel Services*. The system acts as a smart tour operator for composing simple services provided in a local area as, for example, flights, trains, hotels and other tourist attractions. The objective is to provide users with a product, *Travel*, that is the composition and orchestration of atomic services.

The papers is organized as follows: Section II discusses the languages for injecting goals and deploying capabilities into the system. Section III provides details about the dynamic and distributed architecture that emerges for addressing a user-request. Section IV illustrates the core algorithm for allowing the agent to reason on goals and capabilities and for creating plans. Finally, Section V reports some considerations about the approach.

II. Decoupling Goals and Services

MUSA exploits BDI reasoning since it offers the required level of abstraction to build an autonomous and self-aware agent. In particular self-awareness is intended as the ability of agents to recognize their own capabilities (getting knowledge of their preconditions and effects), and to conduct some reasoning over them. The Belief-Desire-Intention (BDI) model was developed at the Stanford Research Institute during the activities of the Rational Agency project [9]. The BDI model assumes software agents had a *mental state* and *a decision making model* representing a promising base for implementing autonomous and self-adaptive systems [10], [11].

In order to make the system able to reason on user-requests and available capabilities a solution is to elect goals and capabilities as first-class entities as it will be described in this section.

A **state of the world** is informally described as a set of non-contradictory first order facts with the assumption that everything that is not explicitly declared is assumed to be false. This is dynamically maintained by the agents of the system as the result of their perceptions and deductions.

A **user-goal** is a desired *change* in the state of the world an actor wants to achieve. In the proposed approach a goal describes the starting state and the final states in terms of states of the world. It is therefore necessary to make a sharp distinction between BDI goals and user-goals. A *user-goal* is injected into the system at run-time (and therefore it not known a-priori by agents) On the other side a *BDI goal* is defined at design-time and the plans for addressing it are hard-coded into the agent. Another difference is that an agent is automatically committed to fulfill all its BDI goals, whereas it owns a higher level of autonomy with respect to user-goals [12]. For example, an agent may check whether it is able to address the goal and then it may decide of committing to it (generally when the agent can get some type of advantage from the situation).

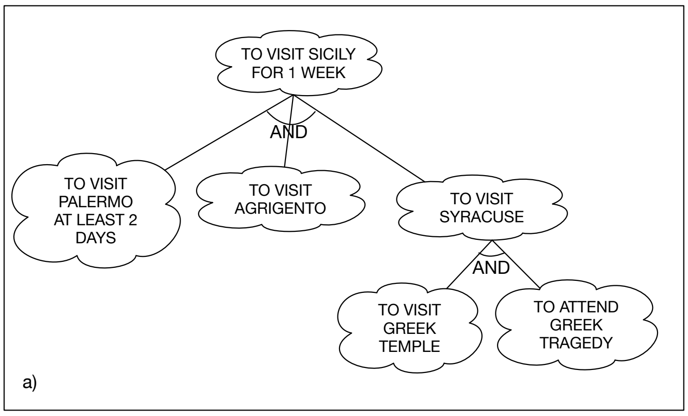
A **goal model** is a directed graph where nodes are goals and edges are AND/OR Refinement or Influence relationships. In a goal model there is exactly one root goal, and there are no refinement cycles. A goal model is an analysts instrument to create dependencies among goals.

A **capability** describes a concrete trajectory in terms of states of the world the system may intentionally use to address a given result. Every agent knows its capabilities together with the way these can be employed. The effect of a capability is an endogenous evolution of the state of the world (a function that takes a state of world and produces a new state of world). The capability can be pursued only if a given pre-condition is true whereas the post-condition must be true after the capability has been successfully executed.

A. GoalSPEC

In order to decouple user-goals from web-services, MUSA provides GoalSPEC [13], a language designed for specifying user-goals and, at the same time, enabling at the same time goal injection and software agent reasoning. It takes inspiration from languages for specifying requirements for adaptation, such as RELAX [14], however GoalSPEC is in line with the definition of goal. The language is based on structured English and it adopts a core grammar with a basic set of keywords that must be extended by plugging-in a domain ontology.

The core element of the metamodel is the *Goal* (desired by some subject). It is composed of a *Trigger Condition* and a *Final State*. The trigger condition is an event that must occur in order to start acting for addressing the goal. The final state is the desired state of the world that must be addressed. The *Subject* describes



```
agent_goal( %TO VISIT SICILY FOR 1 WEEK
    params( [curr_date] , [
       category(curr_date, datetime)]),
    tr_condition( in(curr_date,
       date(2015,7,16),
       date(2015,7,23))),
    final_state( be_visiting(sicily) ),
    herbert
)
...
agent_goal( %TO VISIT SYRACUSE
params( [t] , [
       category(t, integer)]),
    tr_condition( be_at(syracuse) ),
    final_state( and([visited(syracuse,t),
                  t>1] ),
    herbert
)                                          b)
```

Figure 2. On the left side, an example of goal-model the user can inject into the smart travel system. On the right side, an instance of the same goal-model, but expressed in terms of agent's beliefs.

the name of the involved person, role or group of persons that owns the responsibility to address the goal.

In the domain of the Travel Service, GoalSPEC allows the user to describe the kind of travel she desires. Examples of GoalSPEC productions are listed below:

1) WHEN date(16,2,15) THE user SHALL visited(Palermo) OR visited(Catania)

2) WHEN date(DD,MM,YY) AND ($DD > 15$ AND $DD < 20$) THE user SHALL enjoyed(beach)

3) WHEN date(18,2,15) THE user SHALL visited(Syracuse) AND attended(greek_tragedy)

Figure 2.a shows an example of goal-model where each goal must be further refined with GoalSPEC. When injected into the system, the goal is converted into a set of agent's beliefs (Figure 2.b) in which the Trigger Condition and the Final State are expressed as first-order logical conditions to be tested over the current State of the World.

For a complete specification of the syntax of GoalSPEC see [13], whereas details of the conversion into agent's beliefs are provided in [12].

B. Capability

In AI, the need for representing software agent's actions in order to implement reasoning directed towards action is a long-dated issue [15], [16], [17], [18]. An agent is able to achieve a goal by doing an action if either the agent knows what the action is or it knows that doing the action would result in the goal being satisfied [15].

We use a 'robotic-planning-like' approach to address user-goals, in which the Capability is the internal representation of an atomic unit of work that a software agent may use for addressing changes in the state of the world. A Capability is made of two components: an abstract description (a set of beliefs that makes an agent aware of owning the capability and able to reason on its use), and a concrete implementation (a set of plans for executing the job).

We defined a template for providing the abstract description of a capability as a refinement of that presented in [19] for LARKS (language for advertisement and request for knowledge sharing).

Table I
ABSTRACT SPECIFICATION OF THE FLIGHT BOOKING CAPABILITY.

Name	FLIGHT_BOOKING
Input	DPTPLACE : AIRPORT, DPTDATE: DATE, ARRPLACE : AIRPORT, PASSNUM : INTEGER
Output	FLIGHID: STRING, DPTSCHEDULE: DATE, ARRSCHEDULE: DATE
Constraints	$DptPlace \neq ArrPlace$ $DptSchedule > DptDate$ $ArrSchedule > DptSchedule$ $PassNumber > 1$
Pre-Condition	$seat\_available(FlighId, DptSchedule, PassNum)$
Post-Condition	$flight\_booked(FlighId, DptSchedule, PassNum)$
Evolution	$evo = \{remove(being\_at(DptPlace)), add(being\_at(ArrPlace))\}$

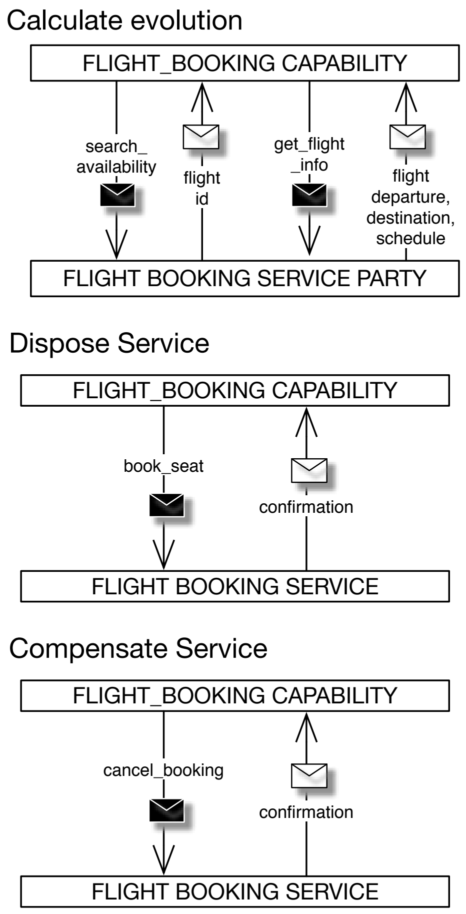


Figure 3. This figure reveals the business logic for the flight booking capability composed of three parts: the calculate evolution, the dispose service and the compensate service.

The template is made of the following elements:

- **Name** is the unique label used to refer to the capability
- **InputParams** is the definition of the input variables necessary for the execution.
- **OutputParams** is the definition of the output variables produced by the execution.
- **Constraints** is an optional (logical or structural) constraints on input/output variables.
- **Pre-Condition** is a condition that must hold in the current state of the world in order to execute the capability.
- **Post-Condition** is a condition that must hold in the final state of the world in order to assert the capability has been profitable.
- **Evolution** is a function that describes how the capability will impact the state of the world as consequence of its execution.

By reasoning on the abstract side (input/output/pre-condition...) the agent may decide when to use the capability. An example of abstract description is provided in Table I.

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 for dealing web-services is made of three parts: the calculate evolution, the dispose service and the compensate service.

The *calculate evolution* protocol is used when composing the whole plan to address a goal; at this stage the agent has to configure the capability for a specific context. This means to establish input/output parameters to generate a contract with other agents it is collaborating with. More details will be in Section IV-B when illustrating the algorithm for the Goal/Capability Deliberation. For instance, the calculate evolution for the flight_booking searches for flights (Figure 3) that are compatible with a given goal, i.e those flights that match with a given DptPlace, DptDate, ArrPlace and PassNumber.

The dispose service and the compensate service protocols will be used for orchestration and self-adaptation purposes (see Section IV-C).

After that a plan has been established for the execution, the orchestration phase generates, through the *dispose service*, the actual value for the user, i.e the user-goal fulfillment. For instance, the dispose protocol for the flight_booking service actually book the specified flight and produces a ticket for the user.

However a plan may be subject to changes for several purposes. The Self-Adaptation Loop monitors for failures or new goals that may affect the running plan. When a plan changes 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.

C. Problem Ontology Description

The previous sections have illustrated how goals and capabilities grounds on the *state of the world* and therefore, under the surface, they employ first-order predicates.

In MUSA the Problem Ontology Diagram (POD) [20] is used to provide a denotation to significant states of the world thus giving a precise semantics to goals and capabilities

An ontology is the specification of a conceptualization made for the purpose of enabling knowledge sharing and reuse [21]. A POD is a conceptual model (and a set of guidelines [22]) used to create an ontological commitment between developers of capabilities and users who inject goals. An ontological commitment is an agreement to use a thesaurus of words in a way that is consistent (even if not complete) with respect to the theory specified by an ontology [23].

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 [22].

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 [20], whereas the link between goals and ontology is detailed in [22]. Finally we also provide GIMT (Goal Identification and Modeling Tool) a tool for supporting ontology building and goal modeling [24].

III. Holonic Architecture for Self-Adaptation

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

Holon is a Greek term for indicating something that is simultaneously a whole and a part [25]. It has been used for introducing a new understanding of ecosystems, and their hierarchical nature. A general definition may be the following:

> A holon is a system (or phenomenon) which is an evolving self-organizing structure, consisting of other holons [26]. 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.

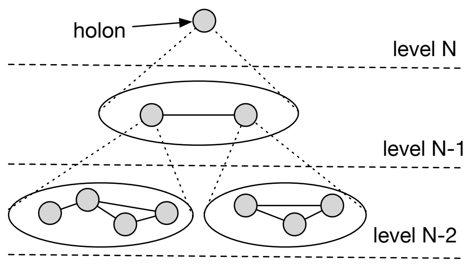


Figure 4. Holarchy layered architecture (elaborated from [27]).

However, each holon is influenced by, and influences its larger wholes. And since a holon also contains parts, it is similarly influenced by, and influences these parts: information flows bidirectionally between smaller and larger systems.

The problem of service composition may be observed as a phenomenon of holon formation [28]. In MUSA services of a choreography maintain their autonomy but they all collaborate for providing an integrated functionality. A composed service is therefore a holon who embeds other component services in a recursive fashion.

A **holonic multi agent system** is a software system made of autonomous holons where the holon is defined recursively (see Figure 4). The holon at generic level N is made of holons a level N-1 kept together by a commitment relationship. The resulting structure is a holarchy, i.e. a hierarchical structure generated by holons and sub-holons where the base case is the agent representing the atomic holon (it is not further decomposable in sub-holons).

In MUSA the commitment function that glues together all the sub-holons to form a super-holon is given by the injected GoalSPEC. The user-goals represent the common objective that the holons have to address.

A. The Basic-Holon Schema

The holon formation is an emergent phenomenon. This section provides details about the static structure of each holon of the system whereas dynamic aspects of holon formation and execution are given in Section IV.

In MUSA all the agents are (atomic) holons, however aggregations of holons (in super-holons) may emerge at run-time for managing composed services. A holarchy is formed as the recursive replication of the same *basic schema*. This template defines that each holon of the system may contain sub-holons playing one of the three roles: service-broker, state-monitor or goal-handler.

The ***service broker*** is the role in charge of establishing a relationship with one or more end points of a remote service. The candidate service-broker owns the capability for managing the conversation with the party that provides the service (for example the flight_booking shown in Figure 3). The service broker must also be able to catch exceptions and failures and to raise the need for self-adaptation.

The ***state monitor*** is the role responsible for monitoring the user environment (both physical and simulated, including persons acting inside). The state of the world is an abstraction for the operative context in which services are going to be provided. The perception of the state of the world is often necessary for invoking services. This role is in charge of providing the service broker with the configuration of input/output parameters for properly invoking a service. It is also responsible of analyzing inconsistencies in the state of the world, due to unfeasible beliefs that could generate service failures.

The ***goal handler*** is responsible for the interpretation of the GoalSPEC and for the recruitment of the service-brokers and state monitors to form a valid holon. The recruitment is based on a procedure called Means-End Reasoning (detailed in Section IV) in which service-brokers and state monitors are selected on the base of capabilities they offer for addressing a desired state of the world. During service execution, this role is also responsible to check the goal life-cycle (active, addressed, failed).

In terms of governance, the goal handler and each service broker and state monitor are simple workers, but at the moment of the holon formation the head of the holon is elected according to a mechanism of trust and reputation (that is out the scope of this paper). The head has three supplementary responsibilities:

1) it is responsible to represent the whole holon to the outside (other holons), thus if service-brokers and state monitors have been selected for a set of capabilities $\delta_1, \delta_2, \ldots$ then the head offers to the outside the composition of these capabilities (namely a *task*) denoted as $\Delta = \langle \delta_1, \delta_2, \ldots \rangle$);
2) it maintains the current state of the world, obtained through the perception of all the sub-holons and it checks for the integrity of the holon structure (verifying all sub-holons are active);
3) even if each worker maintains its autonomy, the head influences their activity i) by guaranteeing their coordination and synchronization, or ii) by deciding for the re-organization of the holon structure as a consequence of failures or unexpected events.

IV. Proactive Means-End Reasoning

This section illustrates dynamic aspects of the formation of holons according to the structural rules specified in Section III-A. The key for dynamically generating holons is what we call Proactive Means-End Reasoning [29], a distributed procedure that allows agents to autonomously decide how to combine their available capabilities and therefore how to generate holons.

A. Goal Model Decomposition and Holon Formation

Section II-A has introduced the language for goal injection. However a goal is rarely injected into the system as an isolated entity. More frequently the user will use more goals to specify its request: a goal model, i.e. a set of correlated goals to be injected at the same time.

Given a goal model (G, R) where $g_{root} \in G$ is the top goal of the hierarchy, the *Goal Model Decomposition* algorithm explores the hierarchy, starting from g_{root} in a top-down fashion. The objective is to trigger the formation of one or more holons able to address the root goal. The algorithm is recursive and it exploits AND/OR decomposition relationships to deduct a goal addressability by observing its sub-goals.

We used δ_j and $\Delta_k = \langle . \rangle$ to respectively denote a single capability and a task. A task is generally provided by a holon. We also introduce $\{.\}$ to indicate a (complete or partial) solution for addressing a generic goal g_i where the 'dot' is a placeholder for a list of tasks that can be alternatively used for the fulfillment of g_i.

An example of solution for a goal has the following form: $\{\langle \delta_1, \delta_2, \ldots, \delta_n \rangle, \langle \overline{\delta_1}, \overline{\delta_2}, \ldots, \overline{\delta_m} \rangle\}$ or the more compact: $\{\Delta_1, \Delta_2\}$ where $\Delta_1 = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$ and $\Delta_2 = \langle \overline{\delta_1}, \overline{\delta_2}, \ldots, \overline{\delta_m} \rangle$. To address the goal the system can alternatively execute Δ_1 or Δ_2.

Figure 5 illustrates how the three roles interact. Once a goal g_i is injected, the goal-handler checks whether the goal is a leaf goal or not. If it is not then the goal-handler proceeds with a top-down recursive decomposition. For

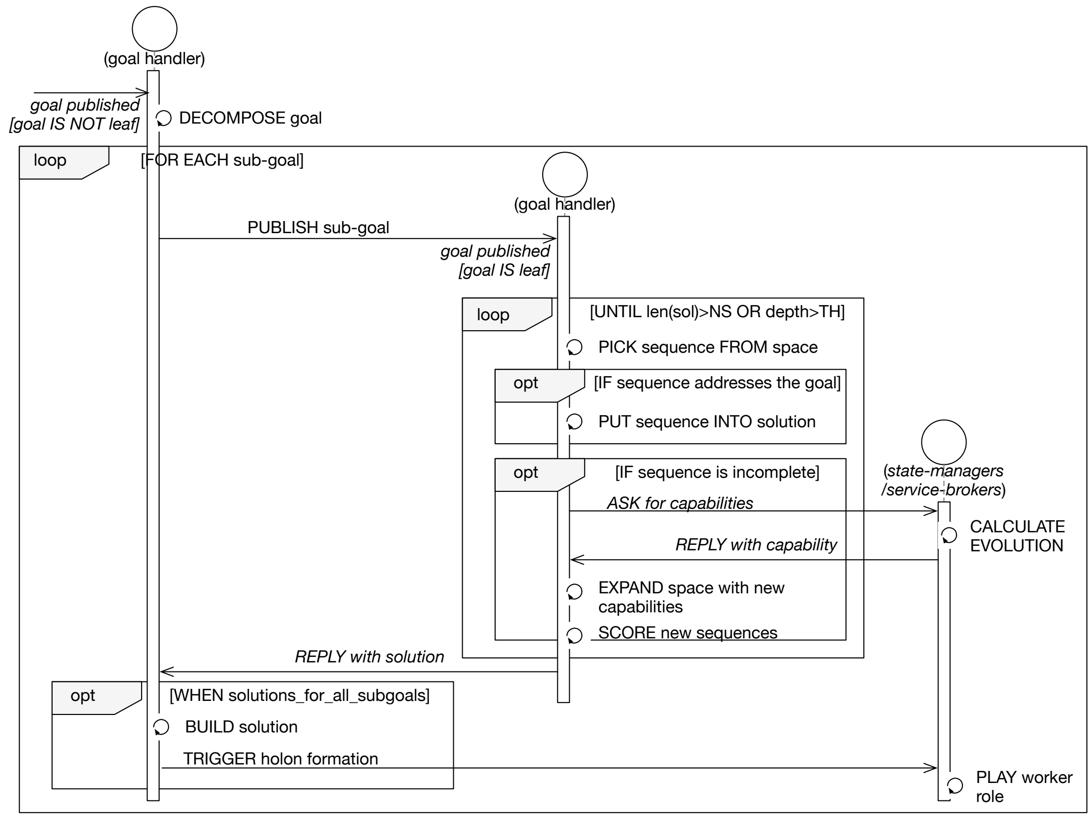

(goal handler)
goal published [goal IS NOT leaf]
DECOMPOSE goal
loop
[FOR EACH sub-goal]
(goal handler)
PUBLISH sub-goal
goal published [goal IS leaf]
loop
[UNTIL len(sol)>NS OR depth>TH]
PICK sequence FROM space
opt
[IF sequence addresses the goal]
PUT sequence INTO solution
opt
[IF sequence is incomplete]
(state-managers /service-brokers)
ASK for capabilities
CALCULATE EVOLUTION
REPLY with capability
EXPAND space with new capabilities
SCORE new sequences
REPLY with solution
opt
[WHEN solutions_for_all_subgoals]
BUILD solution
TRIGGER holon formation
PLAY worker role

Figure 5. Sequence diagrams for goal model decomposition, goal/capability deliberation and holon formation.

each sub-goal, the goal-handler injects again it, marking the relationship to its parent. After that, the goal-handler waits for solutions for all the sub-goals, and then it organizes the composed solution for g_i by triggering the corresponding holon formation.

- If the relationship is an AND decomposition the result is the permutation of all the solutions found for each child node. If g_A is AND-decomposed in two sub-goals g_B and g_C, where $\{\Delta_1, \Delta_2\}$ is the solution of g_B and $\{\Delta_3\}$ is the solution of g_C, then the solution of g_A is $\{\langle\Delta_1, \Delta_3\rangle, \langle\Delta_2, \Delta_3\rangle\}$.
- If the relationship is an OR decomposition the result is the union of all the solutions found for each child node. If g_A is OR-decomposed in two sub-goals g_B and g_C, where $\{\Delta_1, \Delta_2\}$ is the solution of g_B and $\{\Delta_3\}$ is the solution of g_C, then the solution of g_A is $\{\Delta_1, \Delta_2, \Delta_3\}$.

Otherwise, if the goal is a leaf node of the goal model, then the Goal/Capability Deliberation sub-procedure is called.

B. Goal/Capability Deliberation

The Goal/Capability Deliberation algorithm fronts the problem of combining more available capabilities from state-monitors and service-brokers for addressing a goal: given a initial state of the world, a set of capabilities and a goal, the problem is to discover a set of capabilities which composition may address the goal.

MUSA currently adopts an approach based on a search algorithm that simulates the formation of a holon and therefore the execution of various combinations of agent capabilities (see Figure 6). In this phase the service-broker adopts the *calculate evolution* protocol of the capabilities it owns. This protocol allows the agent to customize the capability in order to establish if it can be used for the specific injected goal. For instance the flight_booking capability is customized by searching a flight from a departure place to a destination in a given date/time (see Section II-B). When a complete solution is discovered the algorithm may still continue to search other solutions. It stops after the number of solutions is greater than NS (a system constant) or when time exceeds a threshold and returns all the discovered solutions.

Exploring all possible solutions for addressing a user-goal is a NP-complete problem. The complexity is reduced by putting some constraints during the exploration of the space of solutions. Colored zones of Figure 6 represent invalid solutions that can be discarded. In addition, considering pre/post conditions, only a limited number of capabilities can be exploited at each step of the algorithm, making the execution more scalable and affordable.

The space exploration algorithm compares partial solutions thus to explore firstly the most promising ones, according to the number of sub-goals that are already fulfilled and to a set of domain metrics indicating the global quality of service. The user can specify which domain metrics to consider, for instance the maximum budget or the kind of transportation to use.

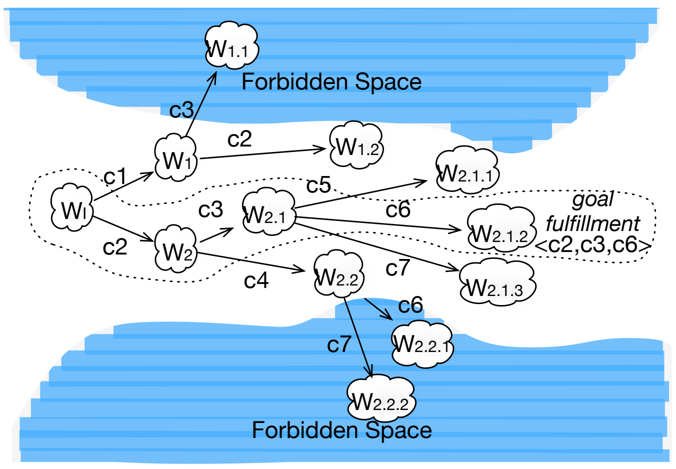


Figure 6. Abstract representation of the strategy used to explore a space of solutions for building a plan.

For a formal description of the Proactive Means-End Reasoning details are provided in [29].

Going back to Figure 5, when one (or more) solutions exist for the root goal of the injected goal model, then one or more holons have been formed for addressing the goal model. One is selected for starting the activities and eventually addressing the goals. The selection of a plan may follow many criteria. For instance in the domain of the travel, the criterion we adopted is the total cost of the travel.

C. The Self-Adaptation Loop

So far the following assumptions holds: 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 system is a distributed and decentralized software, made of a number of autonomous agents, each able to perceive the environment and to act as a broker for some web-services (of which it knows description, end points and business logic); and iii) holons are agents or (temporary) group of agents: each with its own individuality.

Self-adaptiveness is the ability of the whole multi-agent system to dynamically adapt its behavior to the execution context. This is done by each individual agent, through the dynamical execution of its own capabilities, according to a shared plan and to contingent perceptions.

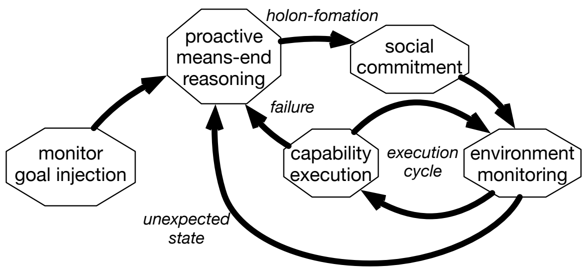


Figure 7. Graphical representation of the Self-Adaptation Cycle.

Orchestration. When a holon is selected for addressing g_{root}, it becomes operative in order to orchestrate all the embedded services and producing the compounded result (see social commitment in Figure 7).

This corresponds to activate its monitoring capabilities and to execute the *dispose service* protocols of service capabilities. State-monitors will be active for the whole real service execution.

In the domain of the *Travel Services*, the service execution has the duration of the user's travel. Service broker will book all necessary flights, hotels, ticket included in the travel plan. The monitor agent, running in the user's mobile device, acquires the user's position, and it may also be used for changing the travel plan at run-time.

Self-Adaptation. Thanks to the work of state-monitors, the head of the holon 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 can not be addressed then the head role of the corresponding sub-holon raises an event of failure, that is cause of an adaptation.

In the running example the adaptation is triggered by the user who modified her goals (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 dis-assembly of the holon and the re-execution of the Proactive Means-End Reasoning procedures but considering the new situation (failures, service availability or new user goals) for generating a new solution.

Before starting the new solution, each holon executes the *compensate* protocols of the capabilities that are no more in the new plan. After that the complete service orchestration activity starts again.

V. Conclusions

Holonic multi-agent systems provide a flexible and reconfigurable architecture to accommodate environment changes and user customization. MUSA is a middleware where the autonomous and proactive collaboration of autonomous agents allows a dynamic (re-)organization of the behavior in order to address user-requests and/or unexpected events. The novelty is that the desired service composition is encapsulated in run-time goal-models that, when injected into the system, trigger a spontaneous emergence of new holarchies devoted to orchestrate the required services.

MUSA have been employed for i) executing dynamic workflows in small/medium enterprises (IDS Project[2]), ii) automatic mash-up of cloud applications (OCCP Project[3], iii) merging protocols for emergency (SIGMA Project[4]).

Authors are already working on 1) a more efficient algorithm for the means-end reasoning, 2) extending the goal language for including uncertainty and norms, 3) an automatic learning approach more robust to ontological discrepancies or language incoherence.

VI. Acknowledgment

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

[2]Innovative Document Sharing: research project funded by the Autonomous Region of Sicily within the Regional Operative Plans (PO-FESR) of the EU Community; 2011-2014.

[3]Open Cloud Computing Platform: research project funded by the Autonomous Region of Sicily within the PO-FESR initiative; 2014-2015 - website: http://aose.pa.icar.cnr.it/OCCP.

[4]Research project for Integrated cloud system of sensors for advanced multi-risk management; 2013-2015.

References

[1] M. Pistore, P. Traverso, M. Paolucci, and M. Wagner, "From software services to a future internet of services." in *Future Internet Assembly*, 2009, pp. 183–192.

[2] S. Staab, W. Van der Aalst, V. R. Benjamins, A. Sheth, J. A. Miller, C. Bussler, A. Maedche, D. Fensel, and D. Gannon, "Web services: been there, done that?" *Intelligent Systems, IEEE*, vol. 18, no. 1, pp. 72–85, 2003.

[3] 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*, vol. 9, no. 1, pp. 1–8, 2007.

[4] L. Sabatucci, C. Lodato, S. Lopes, and M. Cossentino, "Towards self-adaptation and evolution in business process." in *AIBP@ AI* IA*. Citeseer, 2013, pp. 1–10.

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

[6] A. Ricci, M. Piunti, M. Viroli, and A. Omicini, "Environment programming in cartago," in *Multi-Agent Programming:*. Springer, 2009, pp. 259–288.

[7] A. S. Rao, "Agentspeak (l): Bdi agents speak out in a logical computable language," in *Agents Breaking Away*. Springer, 1996, pp. 42–55.

[8] M. E. Bratman, D. J. Israel, and M. E. Pollack, "Plans and resource-bounded practical reasoning," *Computational intelligence*, vol. 4, no. 3, pp. 349–355, 1988.

[9] A. S. Rao, M. P. Georgeff *et al.*, "Bdi agents: From theory to practice." in *ICMAS*, vol. 95, 1995, pp. 312–319.

[10] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, "Reflection, self-awareness and self-healing in openorb," in *Proceedings of the first workshop on Self-healing systems*. ACM, 2002, pp. 9–14.

[11] B. Schmerl and D. Garlan, "Exploiting architectural design knowledge to support self-repairing systems," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. ACM, 2002, pp. 241–248.

[12] L. Sabatucci, M. Cossentino, C. Lodato, S. Lopes, and V. Seidita, "A possible approach for implementing self-awareness in jason." in *EUMAS*. Citeseer, 2013, pp. 68–81.

[13] 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*. Springer, 2013, pp. 235–254.

[14] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 79–88.

[15] R. C. Moore, "Reasoning about knowledge and action," Ph.D. dissertation, Massachusetts Institute of Technology, 1979.

[16] Y. Lesperance, "A formal account of self-knowledge and action." in *IJCAI*. Citeseer, 1989, pp. 868–874.

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

[18] L. Padgham and P. Lambrix, "Formalisations of capabilities for bdi-agents," *Autonomous Agents and Multi-Agent Systems*, vol. 10, no. 3, pp. 249–271, 2005.

[19] K. Sycara, S. Widoff, M. Klusch, and J. Lu, "Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace," *Autonomous agents and multi-agent systems*, vol. 5, no. 2, pp. 173–203, 2002.

[20] 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*, vol. 20, no. 2, pp. 260–304, 2010.

[21] M. Saeki, "Semantic requirements engineering," in *Intentional Perspectives on Information Systems Engineering*. Springer, 2010, pp. 67–82.

[22] 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*, vol. 1099, pp. 66–72, 2013.

[23] N. Guarino, M. Carrara, and P. Giaretta, "Formalizing ontological commitment," in *AAAI*, vol. 94, 1994, pp. 560–567.

[24] 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.

[25] A. Koestler, "The ghost in the machine. 1967," *London: Hutchinson*, 1967.

[26] J. J. Kay and M. Boyle, *Self-organizing, holarchic, open systems (SOHOs)*. Columbia University Press: New York, NY, USA, 2008.

[27] J. Ferber, *Multi-agent systems: an introduction to distributed artificial intelligence*. Addison-Wesley Reading, 1999, vol. 1.

[28] C. Hahn and K. Fischer, "Service composition in holonic multiagent systems: Model-driven choreography and orchestration," in *Holonic and Multi-Agent Systems for Manufacturing*. Springer, 2007, pp. 47–58.

[29] 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.