From Means-End Analysis to Proactive Means-End Reasoning

M. Cossentino L. Sabatucci

Rapporto Tecnico N.:
RT-ICAR-PA-15-01

Febbraio 2015

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

From Means-End Analysis to Proactive Means-End Reasoning

M. Cossentino[1],L. Sabatucci[1]

Rapporto Tecnico N.:
RT-ICAR-PA-15-01

Data:
Febbraio 2015

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo, Viale delle Scienze edificio 11, 90128 Palermo.
[2] Università degli Studi di Palermo, Dipartimento di Ingegneria Chimica Gestionale Informatica e Meccanica, Viale delle Scienze, 90128 Palermo.

From Means-End Analysis to Proactive Means-End Reasoning

Luca Sabatucci and Massimo Cossentino
ICAR-CNR, Palermo
Email: {sabatucci,cossentino}@pa.icar.cnr.it

Abstract—**Self-adaptation is a prominent property for developing complex distributed software systems. Notable approaches to deal with self-adaptation are the runtime goal model artifacts. Goals are generally invariant along the system lifecycle but contain points of variability for allowing the system to decide among many alternative behaviors.**

This work investigates how it is possible to provide goal models at run-time that do not contain tasks, i.e. the description of how to address goals, thus breaking the design-time tie up between Tasks and Goals, generally outcome of a means-end analysis. In this vision the system is up to decide how to combine its available Capabilities: the Proactive Means-End Analysis.

The impact of this research line is to implement a goal-oriented form of self-adaptation where goal models can be injected at runtime. The paper also introduces MUSA, a Middleware for User-driven Service self-Adaptation.

I. Introduction

In the last decade self-adaptation has emerged as a prominent property to tackle some of the most important challenges for developing complex distributed software systems. Self-adapting systems are able to adapt their behavior in response to their perception of the environment and the system itself. As long as software systems grow in size, complexity, and heterogeneity, it becomes central to make them more versatile, flexible, resilient and robust by making them able to dynamically self-adapt to changing environmental conditions.

Self-adaptation has deep roots in several research fields, as for instance, artificial intelligence, biological inspired computing, robotics, requirements/knowledge engineering, control theory and fault-tolerant computing, and so on. Researchers in these areas have investigated different research issues that the term self-adaptation unifies under a common terminology.

Two research roadmaps [1], [2] indicates the contribution that research on software engineering may provide to the topic. In particular up to date, little endeavor has been made to establish approaches for a systematic provision of self-adaptation. In [1] authors agree that the way self-adaptation has to be conceived depends very much on aspects as users, their needs and the characteristics of the environment. Modeling and monitoring these aspects is the key for enabling a software to adapt its behavior.

The system must also maintain a set of high level invariant requirements that indicates the ultimate objective of the system and that drives the adaptation regardless of the environmental changes or uncertainty. To this regards traditional requirements specification languages need to evolve for explicitly encapsulating points of variability in the behavior [3] and elements of uncertainty in the environment [4]. These elements must be first class entities the system can exploit to decide how to act.

In the agent-oriented software engineering (AOSE) research area, one of the most common approaches for facing the emerging challenges posed by self-adaptive software is the use of goal-directed behavior, where the goal is the conceptualization of the objectives the system has to address. For instance, some BDI programming languages explicitly introduce keywords for specifying goals [5].

However, to date, a semantic gap exists between requirement specifications defined at design-time and the concept of goal used at run-time [6]. This represents a limitation especially in the development of self-adaptive and fault-tolerant systems.

A solution has been presented [6] where authors use goal models at runtime and provide an operational semantics for specifying the dynamics of goals, maintaining the flexibility of using different goal types and conditions.

Dalpiaz et al. [7] propose a new type of goal model, called runtime goal model (RGM) which extends the former with annotation about additional state, behavioral and historical information about the fulfillment of goals, for instance explaining when and how many instances of the goals and tasks need to be created.

The novelty of the proposed contribution is to present the concept of *Proactive Means-End Reasoning* as a variation of the classic activity of means-end analysis. The latter represents one of the manual steps of methodologies for modeling goal models. It aims at providing an operationalization of goals i.e. analyzing how to address the desired result specified by a goal. At the best of our knowledge, to date this is a purely human activity. This paper aims to shown that under given assumptions, and simplifications, it is possible to introduce the proactive means-end reasoning as a software agent ability to decide how to address a goal injected at runtime by the user in the system. We exploit this property for building a self-adaptive system in which Goals and Capabilities are two independent entities that may be deployed by different development team. We also developed a prototype called MUSA (Middleware for User-driven Service self-Adaptation) that contains a concrete implementation of the presented conceptual framework.

The paper is structured as follows: Section II presents the theoretical background that introduces the basic concepts. Section 1 introduces the ingredients for the self-adaptation approach that is presented in Section IV. A critical analysis is presented in Section V, and finally some conclusions are drawn in Section VI.

II. Formal Foundation

This section illustrates the theoretical background that introduces the basic concepts of this paper.

A. State of the World Definition

We consider the software system has a (partial) knowledge about the environment in which it runs. The classic way for expressing this property is (Bel a φ) [8] that specifies that a software agent a believes φ is true, where φ is a generic state of affair. We decided to limit the range of φ to first order variable-free statements (facts). They are enough expressive for representing an object of the environment, a particular property of an object or a relationship among two ore more objects. A fact is a statement to which it is possible to assign a truth value. Examples are: $tall(john)$ or $likes(john, music)$.

Definition 1 (Subjective State of the World)**.** *We define the subjective state of the world in a given time t as a set $W^t \subset S$ where S is the set of all the (non-negated) facts $(s_1, s_2 \dots s_n)$ that can be used in a given domain.*

W^t has the following characteristics:

$$W^t = \{s_i \in S | (Bel\ a\ s_i)\} \tag{1}$$

where a is the subjective point of view that believes all facts in W^t are true at time t; and

$$\forall s_i, s_j \in S \text{ if } s_i \wedge s_j \vdash \perp \text{ then } \begin{cases} s_i \in W^t \Rightarrow s_j \notin W^t \\ s_j \in W^t \Rightarrow s_i \notin W^t \end{cases} \tag{2}$$

i.e.: the state of the world is a consistent subset of facts with no (semantics) contradictions.

W^t describes a closed-world in which everything that is not explicitly declared is assumed to be false. An example of W^t is shown in Figure 1, whereas, for instance the set $\{tall(john), small(john)\}$ is not a valid state of world since the two facts produce a semantic contradiction.

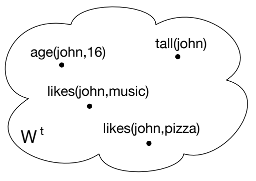

Fig. 1. Example of a State of the World configuration at time *t*.

A Condition of a state of the world is a logic formula composed by predicates or variables, through the standard set of logic connectives ($\neg, \wedge, \vee$). A condition may be tested against a given W^t through the operator of unification.

B. Goal Definition

In many Goal-Oriented requirement engineering methods the definition of *Goal* [3] is: "a goal is a state of affair that an actor wants to achieve". We refined this statement to be compatible with the definition of W^t as: "a goal is a desired *change* in the state of the world an actor wants to achieve", in line with [**?**]. Therefore, to make this definition operative, it is useful to characterize a goal in terms of a triggering condition and a final state.

Definition 2 (Goal)**.** *A goal is a pair: $\langle tc, fs \rangle$ where tc and fs are conditions to evaluate (over a state of the world) respectively when the goal may be actively pursued (tc) and when it is eventually addressed (fs). Moreover, given a W^t we say that*

$$\textit{the goal is active iff } tc(W^t) \wedge \neg fs(W^t) = true$$

$$\textit{the goal is addressed iff } fs(W^t) = true.$$

It is worth noting that when the triggering condition is trivially defined as true, then the above reported definition coincides with the classical definition of Goal.

It follows the definition of goal model, inspired by [7]:

Definition 3 (Goal Model)**.** *A goal model is a directed graph, (G,R) where G is a set of goals (nodes) and R is the set of Refinement and Influence relationships (edges). In a goal model there is exactly one root goal, and there are no refinement cycles.*

Figure 2 is the partial goal model, represented with the *i** notation, for the meeting scheduling case study. This example, redesigned from [7], includes functional (hard) goals only, and AND/OR refinements. The root goal is to provide meeting scheduling services that is decomposed in schedule meetings, send reminders, cancel meetings and running a website. Therefore meetings are scheduled by collecting participant timetables, choosing a schedule and choosing a location. Such a model is useful for analysts to explore alternative ways for fulfilling the root goal.

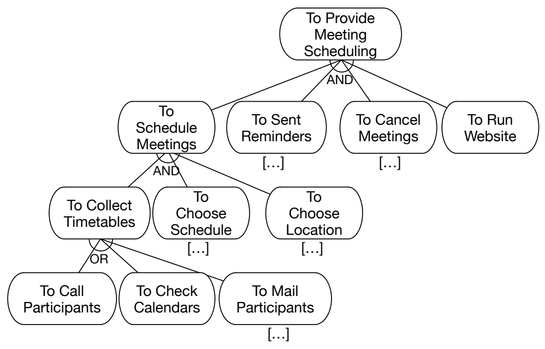

Fig. 2. Portion of Goal Model taken from [7] for the Meeting Scheduling case study. For reasons of space, the tree has been truncated (with respect to the original one) where the symbol [. . .] appears.

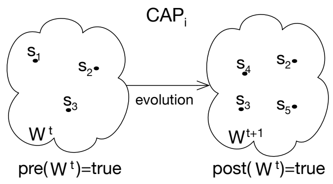

Fig. 3. Graphical representation of how the execution of a system Capability will affect the state of the world.

C. Capability Definition

In many goal-oriented approaches, a Task is the operationalization of a Goal. This means that each task, in a goal model, is associated to one (or more) leaf goal(s). This association is made at design time as the result of a human activity called means-end analysis. In the $i*$ conceptual model [9], a means-end link introduces *a means to attain an end* where *the end can be a goal, task, resource or softgoal, whereas the means is usually a task*. The TROPOS methodology [3] introduces means-end analysis as an activity for identifying (possibly several alternative) tasks to satisfy a goal.

The task is therefore an analysis entity that encapsulates how to address a given goal according to the following statement: "a Task T is a means to a Goal G (G being the end) when one or more executions of T produce a post-situation which satisfies G" [10].

We explicitly introduce the concept of system Capability for introducing a difference between means-end analysis made at design-time and at run-time.

Definition 4 (Capability)**.** *A capability is a run-time property of the system that may be intentionally used to address a given result. The effect of a capability is an endogenous evolution of the state of the world. The evolution is expressed as a function that takes a state of world W^t and produces a new state of world W^{t+1} by manipulating statements in W^t. The evolution can start only if a given pre-condition is true over the current state of the world ($pre(W^t) = true$). If the capability has been successfully executed, then a post-condition must be true in the resulting state of the world ($post(W^{t+1}) = true$).*

A representation of the evolution, preconditions and post-conditions of a capability is shown in Figure 3.

The main difference between Capability and Task is that the former has not an explicit link with any goal. Capability is like a 'tool' the system owns for changing the current state of the world. It is up to the system to execute a reasoning process for establishing which capability or sequence of capabilities to select in order to address a target goal.

Problem 1 (Proactive Means-End Reasoning)**.** *Given a current state of the world W_I, a Goal Model (G, R) and a set of available Capabilities C, the Proactive Means-End Reasoning concerns finding a set of capabilities $CS \subseteq C$ in which each capability will address one of the goals of the Goal Model (G, R), thus to grant the achievement of the root goal g_{root}.*

In the next section, we introduce the role of ontology in our approach and the specific metamodel we refer to in defining the ontological models for our systems.

III. Ontology-based System Self-Awareness

Self-* properties may related to software quality factors as defined in the ISO 9126-1 quality model. In particular there are evidences that self-awareness, considered the base property for all the other self-* system properties, impacts quality factors, such as maintainability, functionality, and portability [11].

Self-awareness is often referred as the ability of a software agent to know (and reason on) its state and its behavior. In Philosophy, the term awareness is often associated with theories of consciousness and of self-referential behavior [12]. "Thinking that One Thinks" resumes a very high level of awareness that is common in human consciousness [13].

In Artificial Intelligence this property is often implemented for enabling a software agent to plan its behavior. The theory of self-knowledge and action [14] asserts an agent achieves a goal by doing some actions if the agent knows what the action is and it knows that doing the action would result in the goal being satisfied [15].

With the aim of implementing self-aware software agents we consider Goals and Capabilities as first-class entities for agent deliberation. The abilities an agent needs to address a goal that is provided at run-time are: 1) to know its own capabilities, their usage and effect and 2) to decide which capability to execute (and in which order) for addressing a desired result.

The aim of the remaining part of this section is to describe how we provide our agents with the proposed self-awareness skills (Section III-D). In order to better detail the approach, before that, we introduce the ingredients needed to achieve our purpose: the way we depict the problem domain using an ontology, a language for specifying goals that refers to ontological elements as keys for grounding the goals on the problem and, finally, a language for declaring capabilities that supports the separation between an abstract description of the capability and its concrete implementation.

A. The Domain Ontology Description

An ontology is a specification of a conceptualization made for the purpose of enabling knowledge sharing and reuse [16]. 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 [17].

A Problem Ontology (PO) [18] is a conceptual model (and a set of guidelines) used to create an ontological commitment for developing complex distributed systems. This artifact aims at visualizing an ontology as a set of concepts, predicates and actions and how these are related to one another.

In this section we exploit the PO for encoding a specific domain of interest as the baseline for implementing system *self-awareness* of Goals and Capabilities.

The metamodel of a PO artifact (Figure 4) has been inspired by the FIPA (Foundation for Intelligent Physical Agents)

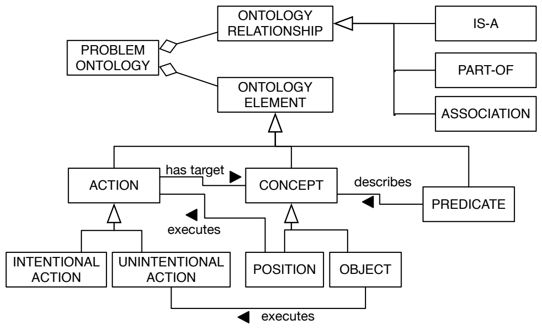

Fig. 4. Metamodel of the PO artifact.

standard [19] and refined for being used in the ASPECS methodology [18], [20]. It is shown in Figure 4 and explained below.

- A ***Concept*** is a general term usually used in a broad sense to identify "*anything about which something is said*" [21] that has a unique meaning in a subject domain. We use the term Concept just for representing classes of domain entities;
- a ***Predicate*** is the expression of a property, a quality or a state of one (ore more) concept(s). It could define a formal structure for statements and rules that relate instances of those concepts;
- an ***Action*** is defined as "*the cause of an event by an acting concept*" (adapted from [22]). Actions are classified as intentional and unintentional [23] where intentionality implies a kind of consciousness to act, whereas Unintentional Action is an automatic response governed by fixed rules or laws;
- a ***Position*** is a specialization of concept performing Actions (both Intentional and Unintentional).
- finally, an ***Object*** represents all the concepts that can perform only unintentional actions.

For what concerns relationships, the PO metamodel supports:

- ***is-a*** (or is-a-subtype-of) that is the relationship that defines which objects are classified by which class, thus creating taxonomies;
- ***part-of*** relationship (or the counterpart *has-part*), in which ontological elements representing the components of something are associated with the ontological element representing the entire assembly;
- ***association*** that is a general purpose relationship for establishing propositions that links two ontological elements. They are particularly useful for defining a formal structure for instances of related concepts.

This representation, mainly human-oriented, is particularly relevant for developing cognitive system that are able of storing, manipulating, reasoning on, and transferring knowledge data directly in this form [20]. As an example, many Belief-Desire-Intention (BDI) [24] system implementations use first-order predicates for describing entities of the environment (and their properties) that can be perceived and manipulated by a software agent.

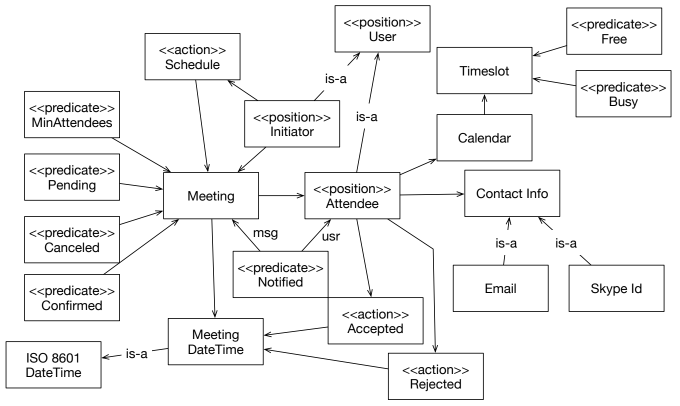

Fig. 5. Example of Problem Ontology for the Meeting Scheduling application context. Ontology elements represented without stereotypes are to be read as concepts by default.

Here let's consider how a requirement analyst uses the ontology diagram to provide a denotation to significant states of the world, thus allowing to give a precise semantics to goals and capabilities. Given that a state of the world is made of statements (that are considered true in a given time instant), then ontology-based propositions are built with a formal structure by grounding on concepts, predicates and actions. Translation of formal ontology into representation systems is a well-known topic in the state of art about knowledge representation [25].

For instance, with reference to the ontology of the meeting scheduling application (Figure 5), a state of the world may have the form shown in Figure 6, where *m123* is an instance of the Meeting concept, *mario.rossi* is an instance of Attendee and so on. At the same way, the statement *notified(m123,mario.rossi)* is an instance of the Notified predicate of the PO.

B. A Goal Specification Language

The GoalSPEC language [26] has been specifically designed for enabling runtime goal injection and software agent reasoning. It takes inspiration from languages for specifying requirements for adaptation, such as RELAX [4], however GoalSPEC is in line with Definition 2. The language is based on structured English and it adopts a core grammar with a

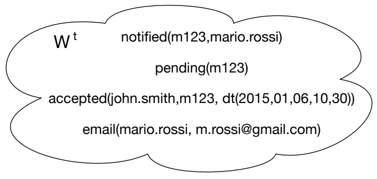
W t
notified(m123,mario.rossi)
pending(m123)
accepted(john.smith,m123, dt(2015,01,06,10,30))
email(mario.rossi, m.rossi@gmail.com)

Fig. 6. A state of the world built in conformance to a domain ontology.

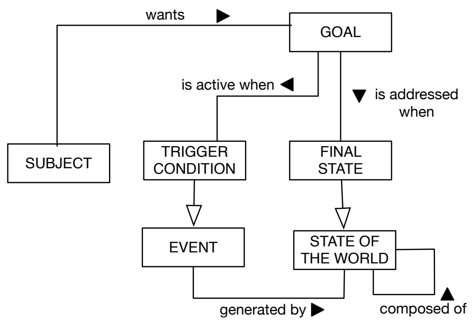

Fig. 7. The Core Metamodel of the Goal Specification Language.

basic set of keywords that must be extended by plugging-in a domain ontology.

As already stated, the core grammar of GoalSPEC is in line with Definition 2. Figure 7 represents its metamodel. The main entity is *Goal* (wanted by some subject) that it is composed of a *Trigger Condition* and a *Final State*. The subject is a noun that describes the name of the involved person, role or group of persons that owns the responsibility to address the goal. 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.

It is worth underlining that both Trigger Conditions and Final States must be expressed by using a State of the World, that in turn is expressed through domain ontology predicates.

For a complete specification of the syntax of GoalSPEC see [26]. Some examples of GoalSPEC productions for the domain of the Meeting Scheduling are listed below:

1) WHEN schedule(Usr,Meeting) THE system SHALL PRODUCE canceled(Meeting) OR confirmed(Meeting)

2) WHEN pending(Meeting) AND meeting_datetime(DT) AND attendee(Meeting,A) THE system SHALL PRODUCE notified(A,Meeting,DT)

3) AFTER 2 days SINCE WHEN notified(Usr,Meeting,DT) THE system SHALL PRODUCE accepted(Usr, Meeting,DT) OR rejected(Usr, Meeting,DT)

Each of the items shown before are goals. For purpose of clarity we put in uppercase the keywords of the language, and in lowercase the domain specific predicates constrained by the problem ontology (Figure 5). Goal 1 indicates that 'when the software agent knows a user is going to schedule a meeting, then it should bring the meeting to a state of canceled or confirmed'. Goal 2 states that 'when a meeting is yet in a state of pending, but a date-time is going to be proposed to a set of attendees, then each of these attendees has to be notified about'. Finally, Goal 3 says that 'when two days past since the notification has been sent, then the system must collect the results (accepted or rejected)'.

After that a set of goals has been completed, it can be injected into the running system, thus to let the system try to address them. We called this mechanism *goal injection* [27].

C. A Capability Specification Language

In AI, the need for representing software agent's actions in order to implement reasoning directed towards action is a long-dated issue [?], [8], [14], [15]. An agent is able to achieve a goal by doing an action if either the agent knows what the action is and knows that doing the action would result in the goal being satisfied [14]. This topic has become even more actual because the amount of services deployed in the web is exponentially growing and researchers are looking for ways for automatically searching, selecting and composing them [28].

We use Capability as an 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 body implementation (a set of plans for executing the job).

Whereas we define a template for providing the abstract description of a capability, we do not provide any language for the body, leaving the choice of the specific technology to the developer. The proposed template (Table I) is a refinement of that presented in [28] for LARKS (language for advertisement and request for knowledge sharing).

TABLE I
TEMPLATE FOR DOCUMENTING A CAPABILITY DESCRIPTION.

Name	Unique label used to refer to the capability
InputParams	Definition of the input variables necessary for the execution.
OutputParams	Definition of the output variables produced by the execution.
Constraints	Optional (logical or structural) constraints on input/output variables.
Pre-Condition	Condition that must hold in the current state of the world in order to execute the capability.
Post-Condition	Condition that must hold in the final state of the world in order to assert the capability has been correctly executed.
Evolution	Function of evolution $evo : W \longrightarrow W$ as described in Section II

Tables II and III are two examples of capabilities that work with emails. The Proposal Mail Sender capability encodes a question into the content of an email, thus the receiver can select two links, for answering yes or not. The second capability, Collect Response, looks at all the received answers to a given question and returns an array in which there is an item for each user who replied.

There is also a special category of capabilities that is *Cloud Capability*. These capabilities have been created for interacting with a REST application on the cloud. An example is the Google Calendar Check capability reported in Table IV. The aim of this capability is to interact with users' google calendar account for obtaining whether a given time slot is free or busy.

TABLE II
ABSTRACT SPECIFICATION OF THE PROPOSAL MAIL SENDER CAPABILITY.

Name	PROPOSAL_MAIL_SENDER
InputParams	QUESTION : TEXT, RESPONSEID: STRING USERMAIL : STRING
OutputParams	NONE
Constraints	$format(UserMail, RFC\_5322\_Address\_Specification)$
Pre-Condition	$email(Usr, UserMail)$
Post-Condition	$notified(Question, Usr)$
Evolution	$evo = \{add(notified(Msg, Usr)), add(mailed(UserMail, Question)) add(questioned(Usr, ResponseId))\}$

TABLE III
ABSTRACT SPECIFICATION OF THE COLLECT RESPONSE CAPABILITY.

Name	COLLECT_MAIL_RESPONSES
InputParams	RESPONSEID : STRING
OutputParams	RESPONSEARRAY : ARRAYOF(RESPONSE(USR,$\{yes \mid not\}$))
Constraints	NONE
Pre-Condition	$questioned(Usr, ResponseId))$
Post-Condition	$accepted(Usr, ResponseId) \lor rejected(Usr, ResponseId)$
Evolution	$evo = \{add(accepted(Usr, ResponseId)) add(rejected(Usr, ResponseId)) remove(questioned(Msg, ResponseId))\}$

D. Implementing Self-Awareness

Reasoning about knowledge and belief is still an issue of concern in philosophy and artificial intelligence. For the purpose of this work, some simplifications have been assumed for aiming at the core of this research problem.

The principle at the base of the approach is that a software agent can store injected goals, its capabilities, the computational state and the execution process by using the same belief baseFirst-order logic provides a well-understood model-theoretic semantics and it enables characterization of reasoning on goals and capabilities in terms of classical notions of deduction and consistency [29].

The issue of implementing injected user-goals into a BDI [24] agent has been already considered in some recent works in literature [30]. Similarly, also annotating agent's capabilities/services with a first-order logic semantics is an open branch of research [5].

Here it is a couple of examples of how respectively Goal 1 and Goal 2 reported in Section III-B may be encoded in a software agent's belief base:

```
agent_goal(
    params( [usr,mtg] , [
       category(usr, attendee),
       category(mtg, meeting) ]),
    tr_condition( schedule(usr,mtg)),
    final_state( or(
        canceled(mtg),
        confirmed(mtg) ) ),
    system
)
agent_goal(
    params( [mtg,dt,a], [
        category(mtg, meeting),
        category(dt, meetingdatetime),
        category(a,attendee) ) ,
    tr_condition( and(
        pending(mtg),
        meeting_datetime(dt),
        attendee(mtg,a) ) ),
    final_state( notified(a,mtg,dt ),
    system
)
```

This code has to be read as follows: the agent knows to own a couple of goals. The first goal is linked to two concepts of the ontology: Attendee and Meeting. It has, as triggering condition, the formula *schedule(usr,meeting)* and, as final state, a logical OR condition between two statements: *canceled(meeting)* and *confirmed(meeting)*. The second goal grounds over three concepts of the domain: Meeting, Meeting-DateTime and Attendee. The goal precondition is the logical AND condition of three elements, whereas the final state is the formula *notified(a,mtg,dt)*.

The first advantage of having goals in the agent belief base is that they can dynamically change during the agent life. Indeed, new goals can be added into the belief-base, or existing goals can be retreat. An injected goal is not automatically committed by the agent through a plan (as it happens in many rule-based systems): goal commitment is the result of agent reasoning.

In a similar encoding style, the agent can also store abstract capabilities. Here a couple of examples of the proposal_mail_sender and collect_mail_responder capabilities, respectively.

TABLE IV
ABSTRACT SPECIFICATION OF THE GOOGLE CALENDAR CHECK CAPABILITY.

Name	GOOGLE_CALENDAR_CHECK
InputParams	SLOT : TIMESLOT, USERCALENDAR : CALENDAR
OutputParams	RESPONSEARRAY : ARRAYOF(SLOT(USR,$\{free \mid busy\}$))
Constraints	$format(Slot, slot(dt(year, month, day, hour, minute), dt(year, month, day, hour, minute)))$
Pre-Condition	$calendar(Usr, UserCalendar)$
Post-Condition	$free(Usr, Timeslot) \lor busy(Usr, Timeslot)$
Evolution	$evo = \{add(notified(Msg, Usr)), add(free(Usr, Timeslot)) add(busy(Usr, Timeslot))\}$

```
agent_capability( proposal_mail_sender,
    in_params([question,response_id,usermail]),
    out_params( [] ),
    precondition( email(user,usermail) ),
    postcondition( notified(question, user) ),
    evolution( [
        add( notified(question, user) ),
        add(mailed(usermail, question) ),
        add( questioned(user,response_id) ) ] )
)
agent_capability( collect_mail_responder,
    in_params( [response_id] ),
    out_params([array(response(user,boolean))]),
    precondition( questioned(user,response_id) ),
    postcondition( or(
        accepted(user,response_id),
        rejected(user,response_id) ) ),
    evolution( [
        add( accepted(user,response_id) ),
        add( rejected(user,response_id) ),
        remove(questioned(user,response_id))])
)
```

This code is the faithful reproduction of information in form of logical predicates, shown in Table II and Table III.

IV. Self-Adaptation as the Result of Agent Deliberation

The principle at the base of this approach for software agent deliberation is that each agent knows to know something.

Let us suppose the software agent a knows W^t (the current state of the world), and a goal g is injected. The agent must be able of understanding if g is already addressed in W^t by evaluating the goal triggering condition and final state, respectively $tc(W^t)$ and $fs(W^t)$. If g is yet to be addressed, then the agent starts considering the opportunity to pursue that and therefore it reasons on the set of capabilities C it owns, and for each $cap_i \in C$ how the generic capability affects W^t. We call Goal/Capability Deliberation the ability of discovering a sequence of $cap_i \in C$ which execution will lead to address g (Problem 1).

The BDI software architecture [24], inspired by human attitudes (beliefs, desires, intentions), is a common model for implementing a goal-directed behavior. The assumption of the BDI model is that computer programs can have a *mental state*. Thus BDI systems are computer programs having computational features that are analogues to beliefs, desires and intentions [24]. BDI software agents offer the required level of abstraction to build an autonomous, self-aware and self-adaptive system. However, the basic BDI approach is not sufficient to implement the Goal/Capability deliberation as it has been described before. It is a common practice in current BDI application to develop collections of plans at design time. The objective of the agent is then solved by pursuing these plans for their execution at run time. This mechanism lacks of flexibility because i) plans are directly connected to goals they address and ii) the attitude of an agent to deal with changing circumstances is dependent on how plans are coded for it by the agent programmer.

A. Proactive Means-End Reasoning

We implement an agent execution framework based on an architecture that lays upon the BDI model; it exploits beliefs for storing the knowledge about W^t, Goals and Capabilities, and it allows the agent to reason on how to achieve goals injected at runtime by using the capabilities they own. This is achieved by means of a Proactive Means-End Reasoning algorithm and a Goal/Capability Deliberation algorithm that are implemented through desires and intentions.

Given a goal model (G, R) where $g_{root} \in G$ is the top goal of the hierarchy, the Proactive Means-End Reasoning algorithm explores the hierarchy, starting from g_{root} in a top-down fashion. The objective is to check the root goal addressability according to available capabilities. The algorithm exploits AND/OR decomposition relationships to deduct a goal addressability according to its subgoals.

Algorithm 1 $meansend\_resoning(GM, g_{target}, W_I, C)$

```
1:  $meansend \leftarrow goal\_cap\_deliberation(W_I, g_{target}, C)$
2:  if $g_{target}$ IS leaf then
3:      $sol\_set \leftarrow meansend$
4:  else if $meansend = \emptyset$ then
5:      $dec\_type \leftarrow get\_decomposition\_type(g_{target}, GM)$
6:      $children \leftarrow get\_subgoals(g_{target}, GM)$
7:      for all $g_i \in children$ do
8:          $sub\_sols \leftarrow meansend\_resoning(GM, g_i, W_I, C)$
9:          if $dec\_type$ == 'and' then
10:             if $sub\_sols \neq \emptyset$ then
11:                 $sol\_set \leftarrow permut(sol\_set, sub\_sols)$
12:             else
13:                 return $\emptyset$
14:             end if
15:         else if $dec\_type$ == 'or' then
16:             $sol\_set \leftarrow union(sol\_set, sub\_sols)$
17:         end if
18:     end for
19: end if
20: return $sol\_set$
```

The first step of the algorithm is to check whether at least one solution exists for addressing the given goal (by using the Goal/Capability Deliberation procedure in Algorithm 2). If the target goal is a leaf goal the adopted solution is the one returned by the sub procedure, otherwise, if no solution has been found, the algorithm proceeds with a top-down recursive approach.

- If the relationship is an AND decomposition the result is the permutation of all the solutions found for each children node.
- If the relationship is an OR decomposition the result is the union of all the solutions found for each children node.

Let us indicate with $\{.\}$ a complete/partial solution for the fulfillment of a goal where the 'dot' is to be replaced by a capability or a sequence of capabilities, expressed in the form $\langle c_1, c_2, \ldots, c_n \rangle$. Therefore, a generic solution_set generated by the algorithm has the following form: $\{\langle c_1, c_2, \ldots, c_n \rangle, \langle \overline{c_1}, \overline{c_2}, \ldots, \overline{c_m} \rangle, \ldots\}$

If a goal g_A is AND-decomposed in two sub-goals g_B and g_C, and the algorithm finds $\{\langle c_1 \rangle, \langle c_2 \rangle\}$ as solutions to g_B and $\{\langle c_3 \rangle\}$ as solution of g_C, then the solution of g_A is $\{\langle c_1, c_3 \rangle\}, \{\langle c_2, c_3 \rangle\}$.

Conversely, if a goal g_A is OR decomposed in two sub-goals g_B and g_C, and the algorithm finds $\{\langle c_1 \rangle, \langle c_2 \rangle\}$ as solutions to g_B and $\{\langle c_3 \rangle\}$ as solution of g_C, then the solution of g_A is $\{\langle c_1 \rangle, \langle c_2 \rangle, \langle c_3 \rangle\}$.

B. Goal/Capability Deliberation

Algorithm 1 is mainly intended for exploring the goal model hierarchy and composing partial solutions into a top level solution that addresses g_{root}. However, the core of Algorithm 1 is the procedure called at the first instruction. The procedure for the Goal/Capability Deliberation is reported in Algorithm 2. It exploits agent's mental states in order to evaluate what happens when capabilities are pursed through a simulation of evolution of possible worlds.

The inputs of the algorithm are the current state of the world W_I and a generic goal $g_i \in G$ of the goal model. At each step the algorithm explores the space of solutions by simulating the employment of one of the available capabilities, thus generating a tree of possible evolution paths of the current state. Each path in the tree represents a sequence of capabilities and their endogenous effects over W_I. An instance of execution of the Algorithm 2 is shown in Figure 8. Let us suppose to be able of representing the solution space as a surface in which each point indicates a different configuration of statements $s_i \in S$. Some areas of this surface are marked as 'forbidden', meaning that those configurations are not valid. Therefore, the algorithm analyzes at each step the most promising path. This is evaluated by considering a score function that measures (i) the distance from the final state and (ii) the quality of the partial configuration.

When an evolution path is selected the algorithm checks whether it addresses the goal g_i: in this case the set of capabilities used to obtain the path is marked as one of the possible solutions. Otherwise the algorithm tries to expand the current path by employing other available capabilities. The output of this algorithm is a list of all the solutions that have been discovered.

The algorithm takes an exponential time to complete. To simplify its execution, we assume of: i) selecting (at each step) only a subset of all possible capabilities to expand the tree, ii) exploring a limited space of solutions in which some areas are forbidden (see Figure 8) and iii) employing domain-specific utility functions to measure the quality of the partial solutions.

C. Self-Adaptation

The proactive means-end reasoning procedure may be the ground for engineering a self-adaptative software system. According the roadmap of self-adaptive systems [1], one of the principles for implementing such a system is to explicitly focus on the 'control loop' as an internal mechanism for controlling the system's dynamic behavior.

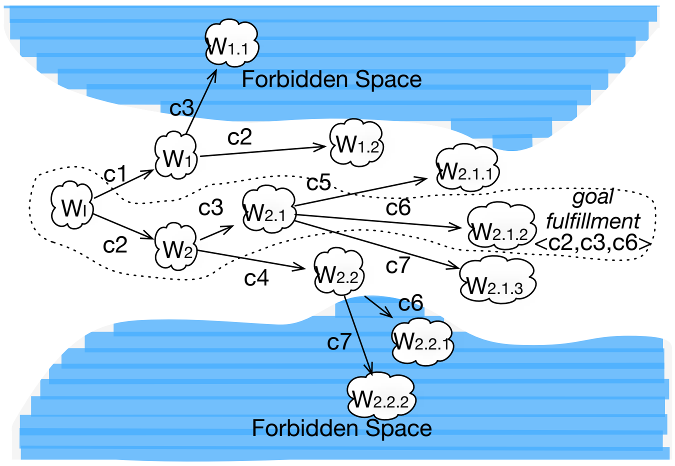

Fig. 8. Abstract representation of the strategy used to explore a space of solutions for building a plan.

Algorithm 2 $goal\_cap\_deliberation(W_I, g, C, Space[=\emptyset])$

```
1: $CS \leftarrow extract\_highest\_scored\_path(Space)$
2: if $g$ is satisfied by $CS$ from $W_I$ then
3:     $sol\_set \leftarrow put(CS, sol\_set)$
4:     if $dim(sol\_set) > max\_dim$ then
5:         return $sol\_set$
6:     end if
7: else
8:     $cap\_set \leftarrow select\_capabilities(C, CS, W_I)$
9:     $Space \leftarrow expand\_and\_score(CS, cap\_set, Space)$
10: end if
11: return $goal\_cap\_deliberation(W_I, g, C, Space)$
```

Figure 9 shows the adaptation cycle whose core activities are: *goal injection*, *proactive means-end reasoning*, *goal commitment*, *environment monitoring*, and *capability execution*. The agent reacts to goal injection by activating the proactive means-end reasoning and by trying to assemble a solution for addressing the goal. If at least one solution is discovered the agent selects the highest scored set of capabilities (according to capability costs, reliability and other QoS factors) for enabling the goal commitment. As a consequence the agent enters in a sub-cycle of monitoring and execution. If everything goes as planned, the goal will be eventually addressed. However, the goal/capability deliberation procedure did not considered exogenous changes of the state of affairs. As a consequence the agent is not ready to act in case of unexpected changes coming from outside the model. When this happens, the proactive means-end reasoning is executed again, but with a different current state of the world. The result will be a new set of capabilities (if possible) for overcoming the unexpected state change. The self-adaptation cycle also considers cases in which the execution of a capability terminates with errors. In this case too, the proactive means-end reasoning is executed, with the shrewdness to mark the capability that failed as ‘unselectable’.

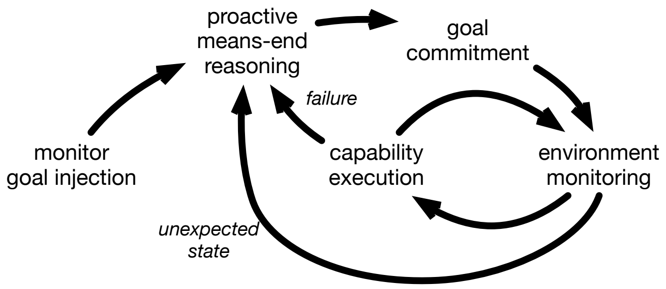

Fig. 9. Graphical representation of the Self-Adaptation Cycle.

V. Discussion and Evaluation

Sections III-IV faced Problem 1 by exploiting a semantic approach for bridging Capabilities and Goals.

This work tries to improve the state of the art in at least two ways. First, the idea of a ***proactive means-end reasoning*** strongly grounds on the research line that explores goal models as mechanisms for software agents to reflect upon their requirements during their operation. In particular, Dalpiaz et al. [7] propose runtime goal model (RGM) that annotate goals with additional information about the fulfillment of goals. Despite RGM is an exceptional instrument for system reasoning, we observed the behavior of the system is wired into tasks of the RGM. The system may adapt its behavior only by selecting (hard-coded) task among alternative OR decomposition relationships. Therefore the research question we raise up is: what if tasks are not provided together with the goal model?

Second, the idea of ***goal injection*** comes from observing that functional requirements could be a runtime entity, to be provided to the system on the need [4]. GoalSPEC is a language for specifying requirements in form of goals for self-adaptive systems. With respect to the work of Whittle et al. [4], GoalSPEC has a simpler syntax but a limited support to uncertainty. The authors have planned of extending GoalSPEC with a new set of keywords for handling uncertainty and high/low priority among goals as future work.

The following subsection introduces the middleware we developed to concretely implement and experiment the above reported theories.

A. A Middleware for User-driven Service self-Adaptation

MUSA (Middleware for User-driven Service Adaptation) is a holonic 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 of dealing with unexpected failures and optimization.

The concept of holon, coined in the field of biology and social science for explaining emergence, equilibrium and self-adaptation, has been recently used in software engineering too [18], [31].

Holon is 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. The main principle that rules an holon is the *Janus effect*, that is a principle of duality: the same entity has its own individuality but at the same time it is made of many parts. 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. An example of abstract holon is a *word* that is part of a sentence, but a whole with regard to the letters that compose it.

IN MUSA holons have been used for bridging self-organization and self-adaptation. A composed service may be seen as a holarchy, i.e. a hierarchy of elements in which each component is a whole and a part at the same time. When developing a complex service, each part maintains its autonomy but it also has to collaborate with other entities for providing a composed functionality. In MUSA this is implemented as a multi-agent system in which elements are able of organizing themselves in holonic structures [32].

Requests for service composition are injected at runtime through the use of user-goal specified in GoalSPEC. Services are deployed in the web as usual, but agents own specific capabilities for dealing with classes of them. Since software agent are deployed in a distributed environment, MUSA implements a distributed version of Algorithm 2 in which the result is not only a set of capabilities for addressing goals, but 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.

The whole system has been implemented by using JASON and CArtAgO. The JASON [33] platform is based on the AgentSpeak language [5] and the BDI theory [24]. 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 which 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, that are simple predicates that state what the agent thinks to be true.

JASON agents are not aware neither of their goals nor of their capabilities. The specification of a goal is strictly connected to the plans to be executed for achieving it. In order to implement self-awareness we use the belief base as illustrated in Section III-D.

CArtAgO (Common ARTifact infrastructure for AGents Open environments) [34] is a general purpose framework/infrastructure that makes it possible to model and program the agents’ environment. In MUSA the body of Capabilities for interacting with web services exploits CArtAgO.

MUSA have been employed in the following research project and case studies:

- *Project IDS* (Innovative Document Sharing) started in 2011-closed in 2014 and funded by the Autonomous Region of Sicily within the Regional Operative Plans (PO-FESR) of the EU Community. MUSA is the core engine for executing dynamic workflows in small/medium enterprises. The architecture includes a BPMN2GOAL component that translates a BPMN 2.0 specification file into a set of GoalSPEC goals. Therefore these goals are injected into the system in which agents are responsible of 1) automatic tasks (as the document classification) 2) to interface with human workers (BPMN Resources) and 3) monitoring manual tasks (as the document supervise) [27].
- *Project OCCP* (Open Cloud Computing Platform) started in 2014-to close in late 2015 and again funded by the Autonomous Region of Sicily within the PO-FESR initiative. MUSA is currently employed for the automatic mash-up of cloud application. The expected result is to allow a user to define a new cloud application as the integration (in terms of data and process) of existing cloud applications.
- *Project PON SIGMA* (integrated cloud system of sensors for advanced multi-risk management) started in 2013-to close in early 2015. This project explores how to merge protocols for emergency when many disasters (for instance earthquake and fire) happen at the same time. MUSA is going to be employed for simulating security operations according to goals and norms.
- Case study on a Smart Travel Agency. This state-of-the-art benchmark has been used for testing the possibility to adopt MUSA in the context of a final user fine configuration of service composition. In this context user goals are indeed used for requesting a fine grained configuration for a 'travel' product. The system is also able to monitor the traveler during its journey and to propose variation to the planned travel when something changes in the context (i.e. a delay or a new user goal).
- Case study on an Exhibition Center system. This case study is still in progress. It aims at testing MUSA in the context of a socio-technical system in which technical aspects are as important as social assets.

B. Limits

This subsection presents a critical analysis of the approach. The first and most relevant topic regards computational complexity. The proposed Algorithm 2 is a search algorithm that improves a breadth-first search even if the time complexity for the worst case is yet $O(b^d)$ where b is the branching factor and d is the depth. This algorithm is a starting point for exploring the research direction. Below we will quickly review some alternative approaches to improve the performance.

Planning/Scheduling algorithms. It could be natural to think that a solution to Problem 1 can be designed by exploiting the state of the art in planning and scheduling. Algorithms for planning are concerned with figuring out what actions need to be carried out for addressing a given result, whereas algorithms for scheduling are concerned with when to carry these actions for the same purpose [35], [36].

SAT solver. Searching in the space of State of Worlds is a combinatorial problem, and therefore it may afforded by a SAT solver. Despite the fact that the Propositional Satisfiability is a NP-complete problem, recently many algorithms (i.e. DPLL, CDCL [37]) reach impressive performances with several hundreds of variables and several thousand of clauses in worst conditions [38].

Case Based reasoning. Case-based reasoning [39] is a problem solving paradigm that in many respects is fundamentally different from other major AI approaches since it is able to utilize the specific knowledge of previously experienced, concrete problem situations. A new problem is solved by finding a similar past case, and reusing it in the new problem situation. This approach would be applied together with a divide and conquer strategy (Algorithm 1) in which a complex problem is decomposed in simpler sub-problems to front them separately.

Another point of discussion concerns the real degree of decoupling between Capabilities and Goals. The authors have introduced the use of an ontology for enabling a semantic compatibility between these two elements. By committing to the same ontology, Capabilities and Goals can be implemented and delivered by different development teams. Our experimental phase has provided evidences that if the ontology is built correctly then the approach works properly. However it must be yet considered how changes in the PO affect the maintenance of Capabilities and Goals. For instance, changing de definition of a predicate in the ontology could have a detrimental impact over the effectiveness of the approach by implying a revision of deployed Capabilities. Authors are already working on building a reasoning algorithm more robust to changes of the language [40], thus to deal with conceptual ambiguities and linguistics flaws (as, for instance similarities and synonyms).

VI. Conclusion

This work is a preliminary step for answering to the research question whether how to implement self-adaptive system in which tasks are not provided together with the goal model. In this vision, system's functional requirements are not hard-coded, but rather they are provided at run-time as Goals. It is necessary an autonomous and proactive software agent able of deciding how to combine its available Capabilities for addressing injected goals. The first result, presented in this work is MUSA, a Middleware for User-driven Service self-Adaptation that implements in a belief-desire-intention programming language an ontology-based algorithm for finding capabilities for addressing a generic goal. The work can evolve in several ways: i) to improve the performance of searching a solution, ii) handling uncertainty and high/low priority among goals iii) to relax the dependency of the approach on an ontological commitment.

REFERENCES

[1] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26.

[2] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[3] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.

[4] 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.

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

[6] M. Morandini, L. Penserini, and A. Perini, "Operational semantics of goal models in adaptive agents," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 129–136.

[7] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos, "Runtime goal models: Keynote," in *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. IEEE, 2013, pp. 1–11.

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

[9] E. Yu, "Modelling strategic relationships for process reengineering," *Social Modeling for Requirements Engineering*, vol. 11, p. 2011, 2011.

[10] R. Guizzardi, X. Franch, and G. Guizzardi, "Applying a foundational ontology to analyze means-end links in the i framework," in *Research Challenges in Information Science (RCIS), 2012 Sixth International Conference on*. IEEE, 2012, pp. 1–11.

[11] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[12] D. M. Rosenthal, *Consciousness and mind*. Oxford University Press Oxford, 2005.

[13] 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.

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

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

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

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

[18] 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.

[19] P. D. O'Brien and R. C. Nicol, "Fipa—towards a standard for software agents," *BT Technology Journal*, vol. 16, no. 3, pp. 51–59, 1998.

[20] 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.

[21] O. Corcho and A. Gómez-Pérez, "A roadmap to ontology specification languages," *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*, pp. 80–96, 2000.

[22] E. Lowe, *A survey of metaphysics*. Oxford University Press Oxford, 2002.

[23] H. Frankfurt, "The problem of action," *American Philosophical Quarterly*, pp. 157–162, 1978.

[24] 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.

[25] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1042814383710083

[26] 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.

[27] 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.

[28] 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.

[29] D. Berardi, M. Grüninger, R. Hull, and S. McIlraith, "Towards a first-order ontology for semantic web services," in *Proceedings of the W3C Workshop on Constraints and Capabilities for Web Services*, 2004.

[30] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, "Goal representation for bdi agent systems," in *Programming multi-agent systems*. Springer, 2005, pp. 44–65.

[31] A. Giret and V. Botti, "Holons and agents," *Journal of Intelligent Manufacturing*, vol. 15, no. 5, pp. 645–659, 2004.

[32] K. Fischer, M. Schillo, and J. Siekmann, "Holonic multiagent systems: A foundation for the organisation of multiagent systems," in *Holonic and Multi-Agent Systems for Manufacturing*. Springer, 2003, pp. 71–80.

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

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

[35] T. L. Dean and S. Kambhampati, "Planning and scheduling." CRC Press, 1997, pp. 614–636.

[36] R. Basseda, M. Kifer, and A. J. Bonner, "Planning with transaction logic," in *Web Reasoning and Rule Systems*. Springer, 2014, pp. 29–44.

[37] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS Press, 2009, vol. 185.

[38] B. Selman, D. G. Mitchell, and H. J. Levesque, "Generating hard satisfiability problems," *Artificial intelligence*, vol. 81, no. 1, pp. 17–29, 1996.

[39] D. B. Leake, "Case-based reasoning," *The knowledge engineering review*, vol. 9, no. 01, pp. 61–64, 1994.

[40] L. Steels, "Language as a complex adaptive system," in *Parallel Problem Solving from Nature PPSN VI*. Springer, 2000, pp. 17–26.