A Possible Approach for Implementing Self-Awareness in JASON

Luca Sabatucci[1], Massimo Cossentino[1], Carmelo Lodato[1], Salvatore Lopes[1], and Valeria Seidita[2],[1]

[1] ICAR-CNR, Palermo, Italy
{sabatucci,cossentino,c.lodato,s.lopes}@pa.icar.cnr.it,
[2] University of Palermo
{valeria.seidita@unipa.it}

Abstract. In Philosophy, the term *awareness* is often associated to theories of consciousness and self-referential behavior. In computer science, the awareness is a topic of increasing relevance in both Software Engineering and Artificial Intelligence, being closely related to autonomy and proactiveness.
We can distinguish two orders of awareness: the first order is the awareness of the environment also known as context-awareness; conversely, self-awareness is a higher order awareness (knowledge about one's own mental states).
Nowadays, many agent oriented languages offer native instruments to implement context-awareness. However, self-awareness is not adequately supported and it requires further considerations. This paper focuses on implementation techniques, based on JASON, for creating software agents able to dynamically reason about their knowledge of the environment, as well as on their missions, capabilities and current execution state.

Keywords: Self-Adaptation, Self-Awareness, Multi-Agent Systems

1 Introduction

Runtime adaptation is, today, an important aspect of complex systems such as distributed systems, defense software, and pervasive computing systems [6]. The most successful examples of self-adaptation in computer science have been designed ad hoc for the specific problem. As a consequence, the experience concerning such systems is hard to generalize, thus making self-adaptation difficult to (a) transfer or reuse in other applications, (b) reason about or analyze to ascertain whether the adaptations will result in proper-running systems, and (c) change.

Recently, the research community in the area of distributed-systems is focusing on the general area of self-* [2, 16]. For instance, self-healing systems attempt to "heal" themselves in the sense of recovering from faults and regaining normative performance levels independently in a way similar to how biological system heals a wound [8]. The lesson learned is that implementing self-adaptation often implies 1) the ability of self-tuning own behavior for reacting to external changes ans 2) the ability of reflecting upon how to behave, own capabilities and resources, for adapting to changes.

Several scientists agree [6] that self-adaptation (for humans as well as for software components) is closely related to the ability of reasoning about its own outer and inner world, or in other worlds, being self-aware. So far, there is still the need for methods that allow to implement these two features by using results from the theoretical approach to the implementation one.

The complexity of current and emerging computing systems has lead to take inspiration from several areas (for instance complex systems, control theory, artificial intelligence, sociology, philosophy, biology, ...). In Philosophy, the term awareness is often associated with theories of consciousness and of self-referential behavior [14, 17]. "Thinking that One Thinks" resumes a very high level of awareness that is common in human consciousness.

Examining the literature we discovered several approaches to self-adaption. Holland and Goodman [11] built an internal model of the environment and generated suitable predictions, Haikonen [9] used the feedback received by a loop in which the model of the environment is implicitly learned in terms of weights of an associative neural network. First Order Logic has been used by Weyhrauch that proposed a system able to make inferences and reflect on them; he established a seminal theory to provide artificial reasoning systems with self reflection capability [18]. The work in [1] deals with the relationship between higher order access and phenomenology through the use of a kernel architecture based on the five axioms of consciousness. Manzotti [12] introduced the externalist point of view, self-conscious ability is reached by continuously comparing the subjective with the objective experience, by reflecting about itself and the world around. So from the theoretical point of view, higher order perception sees an entity able to reflect about itself hence to make inferences about how to act in the world.

This work focuses on creating tools for implementing agents that reason on their own goals, capabilities and knowledge of the environment. Above all, we experienced the use of JASON where an agent is able to reason in order to act in the environment for pursuing its goals but it is not able to reflect on its plans and to re-organize them for dealing with changing situations. Thus inspired by the beliefs of beliefs mechanism we provide some implementation techniques for enabling agents to dynamically reason on their knowledge about the environment as well as on the knowledge of themselves: their goals, their capabilities and their current execution states.

We aim at reducing the gap between self-awareness theory and implementation in the context of multi-agent systems, since self awareness is not native in the most common and used agent oriented language.

The paper is organized as follows: Sections 2 provides an overview on the JASON agent language, Section 3 shows how to implement self-awareness in JASON, section 4 discusses the proposed work using an experiment and finally Section 5 draws some conclusions.

2 The JASON Architecture

The JASON [3] platform is based on the AgentSpeak language [13] and the BDI theory [5]. The Belief Desire Intention (BDI) model was developed at the Stanford Research Institute during the activities of the Rational Agency project. It is derived from the theory of human practical reasoning formulated by the philosopher Michael Bratman. In the BDI model we assume that computer programs can have a *mental state*. Thus, when we refer to a Belief-Desire-Intention system, we are considering computer programs having computational features that are analogues of beliefs, desires and intentions [5].

Beliefs are information the agent owns about the environment where it runs and, of course, it could also be out of date, inaccurate or false.

Desires correspond to the possible states of the world the agent might like to achieve. A desire, however, does not imply that an agent acts for fulfilling it. A desire is then only an option among all the future possible actions of the agent. Instead, intentions are states of world that the agent has decided to achieve. The BDI paradigm is based on a decision making model known as Practical Reasoning (PR). PR consists in two steps. First, the PR process takes into account all the Desires of an agent and selects the most suitable ones according to the agent's Beliefs. This deliberation step produces as results that an agent adopts an intention to pursue the selected desire. Intentions play a much stronger role in influencing the following action of the agent than desires do [4].

Intentions are characterized by an important property: they persist. When an agent adopts an intention, then it attempts to achieve it. If it initially does not success, then it would try again, and it will not easily give up. However, it may happen that beliefs (or other changes in the world) make the intention not useful or interesting anymore. In this case it will be dropped it. Thus the agent reconsiders the new situation and selects another desire.

Moreover, once an agent adopted an intention, its future deliberative reasoning will consider or adopt options that are consistent with that intention. Finally, intentions are closely related to beliefs about the future. In particular, to pursue an intention for achieving a particular state implies that this state is in principle possible. Under normal circumstances, the agent will then succeed with its intention. However, it is possible that its intention might fail. The second step PR is called mean-ends reasoning. It is the process of deciding how to achieve an intention on the base of the actions the agent can perform in his environment. Means-ends reasoning is based on goals or intentions, beliefs and actions; it should generate a plan, that is a sequence of actions. If the mean-ends reasoning works properly it would produce the sequence of action the agent should execute for attaining its goal or intention.

However, it is a common practice in current application to develop collections of partial plans at design time. The task of the agent is then to pursue these plans for their execution at run time. This approach lacks of flexibility because the ability of an agent to deal with changing circumstances is dependent on the plans coded for it by the agent programmer. Nevertheless, this approach can work well in practice and AgentSpeak is based on that model.

AgentSpeak is a programming language based on events and actions [13]. The behavior of agents is ruled by programs written in the AgentSpeak language. 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. Hence, the program of an agent includes a set of base beliefs and a set of plans. Plans are triggered by events and consist of a sequence of actions to be performed. At run-time, an agent can be viewed as consisting of several sets: beliefs, plans, intentions, events, actions and selection functions. Among these elements only beliefs are explicitly represented with modal formulas. Goals or sub-goals are actually considered as the successful final condition of plans.

In JASON, the agent's knowledge is expressed by a symbolic representation (based on implicit ontology) by using beliefs, that are simple predicates (such as *tall(john).* or *likes(john,music).*) that state what the agent thinks to be true. Sources of agent's beliefs are: (i) perception (the agent acquires beliefs as a consequence of sensing the environment), (ii) communication (information acquired from other agents is annotated by beliefs) and (iii) mental activity of an agent, that may generate new information (mental notes), deducted from current beliefs and added to the belief base for convenience. The current state of the agent, its environment, and other agents, can be viewed as its current belief state. States which the agent wants to bring about can be viewed as desires, that are used for activating plans. The context of a plan is used for checking the current situation so as to determine which plan is more suitable to be executed. The context is a logic formula that must be true in order the plan to be applicable. The three parts of a plan are $triggering\_event : context \leftarrow body$. Trigger conditions tell the agent what are the specific changes in the agent's mental attitudes for which the plan is to be used.

A JASON agent is continuously perceiving the environment and it is reasoning about how to act on it for achieving its goals. The reasoning part of the agent's cyclic behavior is done according to the plans the agent has in its own plan library. In the current implementation, JASON agents are actually 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 adaptivity and self-awareness we have to separately define goals and capabilities that is the sequence of actions or of sub-goals necessary for achieving the goal. Such an issue will be discussed in the next sections.

3 Self-Aware Agents in Jason

We consider self-awareness as a sort of ability of introspection the agent can use to adapt its behavior to its own internal state. As well as in BDI systems, when

Listing 1: Beliefs about a goal

```
1  goal(g1).
2  goal_owner(g1,the_system).
3  triggering_condition(g1,g1_tc).
4  event_definition(g1_tc,and( [e1,e2] )).
5  state_true_triggers_event(e1, s1).
6  state_true_triggers_event(e2, s2).
7  property_definition(s1,property(unclassified ,[doc])).
8  property_definition(s2,property(done(receive_document))).
9  final_state(g1,g1_fs).
10 state_definition(g1_fs,and( [s3,s4] )).
11 property_definition(s3,property(classified ,[doc])).
12 property_definition(s4,property(done(classify))).
```

we mention the internal state of an agent, we refer to the agent's knowledge of its Goals, of its Capabilities and of the Environment.

We adopt the definition of system goal from GoalSPEC [15] in which a goal is described as a tuple (Actor, TriggerCondition, FinalState). Where in JASON a goal is a term $!g(t_1, \ldots, t_n)$ which addition/deletion represents a trigger for activating plans, in GoalSPEC, a system goal describes the state of the world the agent wants to achieve. We also refine the concept of Capability from the BDI theory. A Capability is made of 1) the beliefs about *what the agent is able to do*, and plans for actually addressing the declared result.

In the agent belief-base, a goal is represented as a set of beliefs. Listing 1 reports an example of goal. The agent believes that: (at line 1) g1 is a goal; (at line 2) the system is responsible to address g1; (at lines 3-8) the triggering condition of g1 is the AND-condition of two context-properties (*property(unclassified, [doc])* and *property(done, [received _document])*); (at lines 9-12) the final state of g1 is also given by two context-properties ($property(classified, [doc])$ and $property(done, [classify])$).

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 ejected. Differently from JASON, a goal that is injected into the system does not automatically becomes a trigger for a plan. Indeed, when injected, a goal has a lifecycle; Figure 1(a) shows the possible states of a goal. The initial state is *Injected*. When the agent decides to address it, then the goal moves to a state of *Ready*. When the goal's triggering condition is true then the goal becomes *Active*. This state changes when the agent executes its attempt to address it: if the goal's final state is true then the goal is now *Addressed*. Otherwise the goal stays *Active* unless a capability failure occurs.

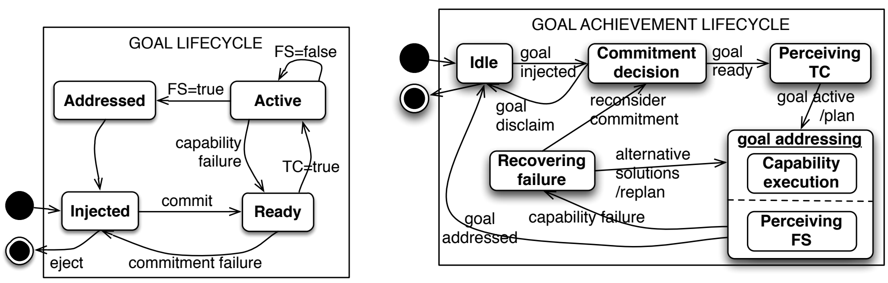
Another advantage is that the agent may reason on goals in the belief base in order to decide its behavior. This property pushes the agent autonomy and proactivity: for instance the agent can decide to commit to a goal only if it finds this action convenient for itself. In any case, the necessary condition since an agent can commit to a goal is shown in Listing 2.

Listing 2: Rule for goal belief-set. The agent is able to commit to a goal when it is able to catch the triggering condition and to address the final state

```
able_to_commit_to( G, Context )
  :-   goal(G)
  & triggering_condition(G, TC )
  & final_state(G, FS )
  & able_to_catch(TC, Context)
  & able_to_address(FS, Context)
  .
```

A Capability is composed of two components: a description and an implementation. The capability description contains indications about when the capability can be used and it is represented as a set of beliefs. On the other hand, the implementation is a set of Jason plans to be executed when the agent wants to activate the capability. Listing 3 reports an example of capability. The agent believes that: (at line 1) *classify_document* is a capability the agent owns; (at lines 2-3) *classify_document* works on objects of type doc; when doc is in the initial condition of *available* this capability promises to return a *classified* doc, if no failures occur. The remaining part of the code concerns the capability implementation. It is composed of three plans: (at line 5) the *prepare* plan is executed before of all the other capability plans; it may be used to prepare data to work on, or to wait some internal condition to be true; (at lines 6-11) the *action* plan contains the body of the capability that addresses the state transition, as a set of traditional Jason instructions; (at line 12) the *terminate* plan is executed after the action is finished and may be used to de-allocate all data that has been used.

All the three plans have the same arguments: the first is the name of the capability and the second is the *Context* variable that will be explained later on this section. This makes to invoke a capability affordable simply by knowing its name. Listing 4 shows the Jason plan to invoke a capability and to retreat it.



(a) The lifecycle of a goal (b) The lifecycle of a capability

Fig. 1: State charts for a goal and its achievement by an agent

Listing 3: Beliefs about a capability

```
agent_capability(classify_doc).
input_state(classify_doc, property(available, [doc])).
output_state(classify_doc, property(classified, [doc]) ).

+!prepare(classify_doc, Context) <- true.
+!action(classify_doc, Context)
    <-
      !get_data(doc, Doc, Context);
      !classify_doc(Doc);
      !register_state(property(classified, [doc]), Context);
    .
+!terminate(classify_doc, Context) <- true.
```

Listing 4: How to invoke a capability

```
+!invoke_capability(Capability, Context)
      :
      agent_capability(Capability)
   <-
      !prepare(Capability, Context);
      !action(Capability, Context);
    .
+!retreat_project_capability(Capability, Context)
    :
      agent_capability(Capability)
   <-
      !terminate(Capability, Context);
    .
```

The advantage of this programming style, is to decouple goals from the plans used to achieve them. However, the agent architecture must include a technique for recreating the link among plans and goals at run-time (means-end reasoning).

Listing 5 is a trivial example of a plan that given a Goal, tries to find a Capability among those the agent owns, that is suitable to address the goal's final state. The plan decomposes the FinalState into a set of properties that must be true for considering the goal satisfied. Then the plan searches in the agent belief base if at least one capability matches with this final state. Of course this plan can be more sophisticated, for instance it could consider additional parameters for selecting the best capability to pick according to non functional requirements such as quality of service, cost, etc.

The variable Context, shown in Listings 3-Listing 4, is a reference for grouping all the beliefs about the environment. There are cases in which an agent may work with different contexts at the same time. It is more common that an unique context exists for each agent. The Context reference may be used to store property of the environment that are believed true, or value of parameters. Listing 6 reports an example to store a property and one data-value into the context. Of course these beliefs can be improved with other parameters on purpose, for instance the trustability of the source, the timestamp of acquisition and so on.

Listing 5: Plans for creating the bridge between a goal to address and the plan to be executed

```
+!link_goal_to_capacity(G, Capability)
  :
        final_state(G, FinalState )
  <-
    !decompose_substate(FinalState , StateList);
    !pick_capability_for(Capability , StateList);
  .
```

Listing 6: Beliefs for representing properties and data of the context

```
environment(context , property(classified ,[doc]).
environment(context , value(doc, 12)).
```

As we already described, the goal trigger_condition belief set declares the requirement for addressing a goal, and the capability_input_state beliefs declare the requirements for activating a capability. Therefore before invoking the capability, it is necessary to check if the current Context variable contains the proper beliefs.

Figure 1(b) shows the lifecycle of an agent when tries to address a goal by using its own capabilities. The agent is initially in an *Idle* state, until a goal is injected. Then it decides if making a commitment to the goal. The trivial case for that is to check if it is able to do it according to the rule reported in Listing 2. Of course this decision may be more complex, for instance by involving additional parameters, and/or by looking for collaborations with other agents in the society. If the agent commits to the goal, then it activates its perception capabilities to detect when the goal's trigger condition is true. When this happens the agent picks a capability according to the plan in Listing 5 and activates this capability according to the plan in Listing 4. During the execution of the capability, the agents also activates its perception capabilities again, in order to check if the final state has been properly addressed, thus to come back to the Idle state. Otherwise the agent declares a capability failure and it enters in a Recovering state. This can be solved in two ways: by selecting another capability to execute, if available, or by coming back to the Commitment state, for changing its intentions. In the case of a collaborative solution, the agent may ask for help from other agents, or it may conclude with the complete failure in addressing the goal.

4 Experiencing self-awareness in workflows

The Innovative Document Sharing (IDS) research project arose to deal with self-adaption in workflow enactment. The mission of the project was to develop a smart document management system to deploy in local small/medium enterprise.

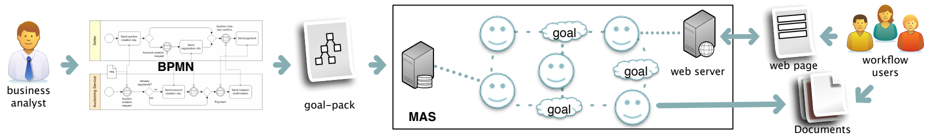


Fig. 2: Architecture of the workflow engine developed for the IDS project.

Currently, a beta version of the described framework is already redeployed for beta testing at three project partners.

The grounding principle of our framework is to decouple *what* should be addressed from *how* this result can be achieved; this allows to make the system and the goal-set evolve independently. Thus, being dynamic, system requirements may be considered as part of the execution context.

The result has been a workflow engine (see Figure 2) realized as a multi-agent system that exploits its features (mainly autonomy and proactivity) in order to monitor the execution state of the process and to discover a distributed solution to unpredictable situations or to specifications' evolution.

In such a system, each autonomous software agent must be self-aware as described in Section 3. However, an agent is not generally able to execute the whole workflow alone, so agents have to collaborate for building a distributed plan.

All together, the agents have to self-organize in groups that proactively discover a distributed solution as the orchestration of many capabilities.

4.1 From BPMN to System Goals Achievement

The whole engine grounds on *GoalSPEC* [15], a language for representing business and system goals. A goal has a responsible actor, and it describes a desired state transition from a triggering condition to a final state. We use the following notation for goal state transition: $g : tc- > fs$. In order to reduce any additional burden for business analysts, we maintain the BPMN as the main interface to model workflows. Hence, we developed *BPMN2Goal*, a component that is responsible to take a BPMN 2.0 XML file in input and to generate as output a set of *GoalSPEC* goals. All the agents of the system are able to interpret *GoalSPEC* and to convert goals into belief-sets (as well as in Listing 1).

A difference with the engine described in Section 3 is that goals coming from the same workflow are correlated. They must be considered as a *goal-pack* that is injected into the system as a whole. Agents collaborate to discover a solution that is a set of potential commitment to a subset of the goal-pack. The goals of a solution are closely related, since if the solution is actuated, then the commitment must be done for all or none of them at the same time.

Listing 7: Example of a perception capability

```
agent_capability(monitor_repository)[category(perception)].
output_state(monitor_repository,property(available,[doc])).
output_value(monitor_repository,value(doc)).

+!prepare(monitor_repository, Context) <- true.
+!percept(monitor_repository, Context)
    <-
      //MONITOR NEW DOCUMENTS
    .
+!terminate(monitor_repository, Context) <- true.
```

4.2 The Agent Level

The workflow enactment is possible because every agent owns special capabilities for addressing the activities specified in the business plan. Anyway agents are not designed with a priori knowledge about which plan is good for a given BPMN activity, nor which is the right order for executing plans.

It is up to the agents' autonomy to generate, at run-time, the missing link between a goal set address and the plans to execute. In line with Figure **??** and Listings 4-5, these are the main features of an agent: (i) agents are able to interpret the injected goals and to reason on the trigger condition and the desired final state; (ii) agents are aware of their own capabilities and can check if they are able to commit to a system goal or a portion of it; and (iii) agents monitor the current state of the context in order to identify when an expected condition holds (trigger condition or final state) and to check if something wrong happens.

Therefore, the architecture of Section 3 provides the engine for allowing an agent to know its capabilities and for creating the bridge between capabilities and goals (what an agent is able to do and what it is expected).

The context is the portion of the environment in which the workflow operates. It may include both software abstractions (document repository, web-services, software resources, database, etc) and physical resources (hardware devices/resources, physical environment and also human participants). In particular we decided to use a different *Context* variable for each workflow instance in order to separately store the state of parallel processes. Moreover the *Context* is shared among all the agents that are involved in the enactment of the workflow instance. Finally, all the beliefs concerning the state of the process and the availability of resources are persistent, thus to let the system to be restored after a restart.

For updating the *Context*, we implemented a proactive monitor loop based on a set of *Perceptions*. Compatibly with the architecture of Section 3, agent perceptions are also capabilities, that are specialized in monitoring a portion of the environment. Listing 7 is an example of perception capability. The *agent_capability* belief uses the Jason Annotation syntax (e.g. *[category(perception)]*) to tag the capability as a perception one. The invocation of perception capability is similar

Listing 8: Example of a capability for interacting with human participants by web pages

```
agent_capability(doc_supervise)[category(manual)].
participant(doc_supervise,manager).
input_state(doc_supervise,property(refined,[doc])).
output_state(doc_supervise,property(approved,[doc])).
output_state(doc_supervise,property(incomplete,[doc])).
output_state(doc_supervise,property(rejected,[doc])).
http_url(doc_supervise,"supervise_doc").


+!prepare(doc_supervise, Context) <- true.
+!serve_page(doc_supervise, Context, RelPath, ParamSet, ReplyHtml)
  :
    RelPath = "supervise_doc"
  <-
    //GENERATE THE HTML PAGE TO REPLY
  .
+!terminate(document_supervise, Context) <- true.
```

to those shown in Listing 4, but it invokes the *percept* plan cyclically until the *Context* variable does not contain the conditions declared in the *output_state*.

Another new category of capability is used to interact with human participants of the workflow. In our framework interaction has been realized by dynamic web pages. Any participant has a personal page that lists all the Tasks that are assigned to him. Any item in this list is a web-link to a dynamic web-page that is generated ad hoc by the agent of the system that is responsible of monitoring the Task progress. In other words the agent acts as a web server, reacting to URL request by building and then routing the proper HTML page. Listing 8 is an example of capability for interacting with human participants. The annotation (*[category(manual)]*) indicates the capability can be selected for handling manual tasks. A new belief *participant* specifies the human role that is involved. The beliefs *input_state* and *output_state* specify as usual the input and output conditions for the capability execution. Finally a new belief *http_url* specifies routing information for the http protocol.

The capability invocation mechanism differs slightly from that of Listing 4. The plan *serve_page* substitutes the plan *action* that is invoked only when a HTTP request incomes from the user. Listings 9 shows three new plans that substitute the plan invoke_capability of Listing 4, and the plan *link_goal_to_capacity* of Listing 5. The event *+request* triggers when a new http request incomes from the user. A http request is associated to a relative path and a set of parameters (in the form `www.mydomain.net/path?par1=val1&par2=val2`). By the *http_url* belief it is possible to identify the capability that is responsible of building the reply. The plan *check_capability_is_ready* verifies the capability is currently active (otherwise the agent will reply with an http error code). Finally the plan *serve_request* invokes the capability and then sends the web-page as response to the user.

Listing 9: How to invoke a capability for handling manual task implemented as a web page

```
+request(Path,ParamSet)[source(ProxyServer)]
  <-
    ?http_url(Capability,Path);
    !check_capability_is_ready(Capability,Path,ParamSet,ProxyServer);
  .
+!check_capability_is_ready(Capability,Path,ParamSet,Proxy)
  :
    ready_for_reply(Capability, Context)
  & ParamSet = paramset(Context,User,Params)
  <-
    !serve_request(Capability, Request, Context);
  .
+!serve_request(Capability, Request, Context)
  :
    Request = request(Path,ParamSet,Proxy)
  & ParamSet = paramset(Session,User,Params)
  <-
    !serve_page(Capability,Context,Path,ParamSet,ReplyHtml);
    .send(ProxyServer, tell, html(Path, ParamSet, ReplyHtml));
  .
```

4.3 The System Level

We already mentioned goal commitment becomes a social activity. This requires to add further Jason plans to the basic mechanism of goal commitment presented in Section 3.

When a new goal-pack is injected, each agent may check if its own capabilities allow to commit to some of these goals. Anyway, commitment is regulated because: 1) goals in the same goal-pack are related one with the others, thus the commitment must be for all or none of them at the same time; 2) more agents could be able to commit the same goal, thus a synchronization technique is required.

The collective commitment to a goal-set is possible thanks to the formation of many *Teams* of agents, each of them proposing a possible *Solution* to the goal-pack. A Solution is a set of related potential commitments to a subset of the goals in the goal-pack that allows to address the whole workflow. Agents involved in a solution form a Team. We omit to report the algorithm used for Team formation and Solution discovery. Eventually, all the Solutions are compared on the base of many possible parameters thus to select the winning one. The agent who formed the winning Team is nominated leader and it is responsible to coordinate the team for actuating the solution. Listing 10 reports the source code for handling the social commitment. The *Solution* variable is a structure that contains the name of the *Leader* agent and a list of *Contracts*. Every contract is a tuple of: 1) the *Agent* who has engaged, 2) the *Goal* to which commit to, and 3) the *Cost* requested by the agent for address the goal (this variable can in turn be a structure formed by multiple parameters). By the *.send* instruction, all the agents in the team receive the notification to commit to the proper goal.

Listing 10: Plan for handling the commitment of a while goal-pack

```
+!social_commitment(Solution ,Pack,Context)
  :
    Solution = solution(Leader,contracts(ContractList))
  <-
    Team = team(Leader, Pack, ContractList, Context);
    for ( .member(contract(Agent, Goal , Cost), ContractList) ) {
      .send(Agent,tell ,hired(Pack, Goal, Team));
    }

    !!monitor_solution_is_complete(Pack,Context);
  .
```

Therefore the leader has also the additional responsibility to monitor when the whole state transition is completed (the workflow is completed).

5 Conclusions

The presented framework is based on decoupling the two concepts Goal and Capability thus to make them evolve independently. To let agent a means-end reasoning, we follow the approach of bridging at run-time *what to do* with *how to do*. We have provided our agents the ability to explicitly reason on their goals, capabilities and the knowledge of the environment.

Another possible approach for implementing a mechanism of self-awareness in JASON is to use the flexible technique of subclassing. By customizing the agent class and the architecture class it is also possible to extend the agent reasoning cycle for implementing a more sophisticated technique for selecting the plan to execute. However, making explicit the deliberation cycle without modifying the agent architecture, (as in our approach) makes the concepts such as Goal, Capability and Context first class citizens of the language together with native concepts of rules, plans and beliefs. The result is a flexible and easy way for customizing the agent architecture but maintaining the compatibility with all the Jason extensions and plugins.

In addition the approach offers several extension points for programmer in order to extend the basic functionalities. An example of this process has been shown in Section 4 in which the Context variable is shared among many agents, the commitment is a social activity, and more categories of capabilities have been introduced over the existing architecture.

Concluding, our approach mainly exploits first-order logic programming that is a common feature of other agent programming languages such as GOAL [10] and 2APL [7]. As well as JASON, these languages share a declarative approach for develop agents that is based on a knowledge base, logic rules and procedural plans (or capabilities in 2APL). However, the deliberation cycle of both GOAL and 2APL is embedded in the respective framework and is not accessible by language interpreter. We suppose our work cold be useful also for these agent programming languages.

Acknowledgements This work has been partially funded by the Innovative Document Sharing (IDS) Project funded by the Autonomous Region of Sicily (PO FESR Sicilia 2007-2013).

References

1. I. Aleksander and H. Morton. Computational studies of consciousness. *Progress in Brain Research*, 168:77–93, 2007.
2. 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*, pages 9–14. ACM, 2002.
3. R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2007.
4. M. E. Bratman. What is intention. *Intentions in communication*, pages 15–32, 1990.
5. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.
6. B. 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. *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.
7. M. Dastani. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008.
8. D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systems—survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.
9. P. Haikonen. *The Cognitive Approach to Conscious Machines.* Imprint Academic, Exeter, UK, 2003.
10. K. V. Hindriks, F. S. de Boer, W. Van Der Hoek, and J.-J. C. Meyer. Agent programming with declarative goals. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 228–243. Springer, 2001.
11. O. Holland and R. Goodman. Robots with internal models - a route to machine consciousness? *Journal of Consciousness Studies*, 10(4 -5):77 – 109, 2003.
12. R. Manzotti. A process oriented view of conscious perception. *Journal of Consciousness Studies*, 13(6):7 – 41, 2006.
13. A. S. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996.
14. D. M. Rosenthal. *Consciousness and mind.* Oxford University Press Oxford, 2005.
15. 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. LNCS 8245.*, In Printing.
16. B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *proceedings of the 14th SEKE*, pages 241–248. ACM, 2002.
17. V. E. Silva Souza. *Requirements-based Software System Adaptation.* PhD thesis, University of Trento, 2012.
18. R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1-2):133–170, 1980.