Abstract—In this paper we propose and implement a modularization framework for Jason that enables developers to decompose agents into separate code units called modules, and by fulfilling an agent-module design contract to conceive agents behaviour design into different levels of abstraction – from a software engineering perspective. Thus, we promote code reuse as well as facilitate independent development, maintainability and extensibility. For our implementation we use annotations mechanism and customizable components in Jason.

I. INTRODUCTION

It is well known that agent-oriented programming languages offer higher level abstraction constructs that facilitate to software engineers the design of computational system solutions, and because of its fundamental principles of autonomy, reactivity and proactiveness, agent-oriented systems have proved to be highly effective dealing with heterogeneous scenarios as well. A good example of this is Jason[1], an interpreter of an extended version of AgentSpeak[9], a high-level agent-oriented programming language which fit in the denominated BDI (Belief-Desire-Intention) model of agency. However, its practical use from industry and developers community to build large-scale software applications still uncertain and lagged by other platforms, e.g. Java, an oriented-object-programming language which provides lower-level-abstraction constructs to model systems than agency approaches, and yet its use stills more extended. This last suggests that Jason lacks of robust and practical mechanisms promoting some software engineering fundamental principles such as code reuse, extensibility and maintainability when building complex agents. That is, from a global system perspective, agents are an atomic component of code reuse, so coding and maintaining complex agents becomes an arduous task, and it will be desirable to decompose agents programs into separate reusable code units. Modularity principle applied to agents points to be a suitable option to deal with this last issue. Thus special attention must be paid to the foundation of mechanisms that allow to structure a single Jason agent program into separate modules, each encapsulating language components such as beliefs, goals, and plans, that together can integrate a particular functionality, and where such mechanisms follow software engineering principles such as low-coupling, high-cohesion, and information hiding.

Within the BDI agency paradigm there exist some papers supporting the idea of modular agents. Regarding the strategy each proposal adopts to tackle agent modularity, it is possible to classify them into three general groups: i) [6],[4] Use the capability concept to encapsulate agent components that together conform a specific agent capability/functionality, and moreover [8], [2] and [3] present their corresponding implementation in Jason+ [5], Jadex and Jack respectively; ii) GOAL [7], designates each module with a mental state, and when this is satisfied, the module becomes the agent focus of execution, this report also presents a notion for action selection; and iii) [10] Associate each module to a specific goal for designating which module should be executed in order to achieve a goal.

On the present paper we propose and implement a modularization framework oriented to allow programmers i) to decompose Jason agents programs; and ii) to dynamically extend agents capabilities\footnote{We use here the term capability as the feature of an agent to react to a determined goal achievement, more specifically, we say that a Jason agent has a capability if has an applicable plan to deal with an event like \(+!x\).}. We achieve that by using the encapsulation principle, which dictates that encapsulated code units must work as a black box, hiding all internal functioning while a public interface must be designated to allow its used for others. So, closer to the first cited group, our modularization approach aims to encapsulate into separate code units, Jason agent components, which together can integrate a set of capabilities and that can be loaded and reused by any agent to handle specific situations or tasks. And is not only the agent which loads and reuse modules components, but also modules can transparently\footnote{Regardless their original source, e.g. artefacts, packages, agents or even other modules.} reuse already available capabilities present in agents. Our perspective is original in the way that, by accepting an agent-module design contract, specified by each module, an agent can know both public module components and those components that module needs from agent. This before leads to a lower agent-module coupling and confers programmers the possibility to decompose agents design not only into separated units but also in different levels of abstraction (from a software engineering perspective), that is to hierarchize agents behaviour programming through modules.

In the present work we use the same notation that [1] to expose our semantics, and in order to implement JasMo framework we make use of source annotation mechanism and customizable components, and thus to keep JasMo agents completely compatible with Jason agents.

The rest of this paper is organized as follows: Section 2 is a breve resume of the general syntax and operational semantics
of Jason, as well as its customizable components described in [1]. Section 3 describes our modularity approach. Section 4 presents JasMo syntax and semantics. Section 5 explains substantial details of our implementation. Section 6 is a small example to illustrate the use of the framework. And finally, section 7 resumes our conclusions and proposes some research lines as future work.

II. JASON

Jason is a Java-based implementation of an extended version of AgentSpeak(L)[9]. An agent program in Jason is composed basically by a set of beliefs and a set of plans, defined by the following grammar:

\[
ag ::= bs \ ps
\]

\[
bs ::= b_1 \ldots b_n \ (n \geq 0)
\]

\[
ps ::= p_1 \ldots p_n \ (n > 0)
\]

\[
p ::= te ct \leftarrow h
\]

\[
te ::= +at | -at | +g | -g
\]

\[
ct ::= ct_1 \top
\]

\[
ct_1 ::= at | -at ct_1 \land ct_1
\]

\[
h ::= h_1 ; \top | \top
\]

\[
h_1 ::= a | g | u | h_1 ; h_1
\]

\[
at ::= P(t_1, \ldots, t_n) \ (n \geq 0)
\]

\[
| P(t_1, \ldots, t_n)[s_1, \ldots, s_m] \ (n \geq 0, m > 0)
\]

\[
s ::= percept \mid self \mid id
\]

\[
a ::= A(t_1, \ldots, t_n) \ (n \geq 0)
\]

\[
g ::= !at \mid ?at
\]

\[
u ::= +b | -at
\]

Here, \(ag\) is used to denote an agent, which is formed by a set of beliefs \(bs\) and a set of plans \(ps\). Thus \(b\) denotes an individual belief, which is a ground (first-order) atomic formula, and \(at\) an atomic formulanec which might be not ground; \(p\) is used for an individual plan, \(te\) for a triggering event, \(ct\) for a plan context, \(h\) for a plan body (\(\top\) indicates an empty plan body), \(s\) for the information source (\(id\) ranges over labels representing the agents in the system), \(a\) for actions, \(g\) for goals, and \(u\) for belief updates which are essentially changes in the agent’s ‘mental notes’.

The reasoning cycle explained next is taken from [8]: during execution, Jason first processes any events and updates the belief base. The interpreter then selects a single event to process and matches it against the plan library to select one or more plans to handle the event. Of these plans, a single plan is then selected to become an intention. Finally, one of the currently active intentions is selected and allowed to perform an action, before the cycle repeats.

Also, Jason implementation adopts an extensibility based approach. It is possible to extend: i) the set of internal actions\(^3\) to be used by the agents; ii) as well as customise and extend different components of the interpreter as agent class, which determines features such as the belief-revision and selection functions used in the reasoning cycle; iii) the agent architecture reddefining how the agent interacts with the external world; iv) belief base, the way stores and organize the set of beliefs; and v) user-defined pre-processing directives that can be useful to extend parsing.

III. JasMo – A Modularity Approach in Jason

In simple words, we conceive a JasMo module as encapsulated Jason code, plus a header stipulating an use design contract conformed by: i) an habilitation list, as a set of predicate indicators referencing components which represent capabilities that module offers to an agent as an extension to its own set of capabilities; and ii) an requirement list, specifying a set of predicate indicators referencing capabilities expected to be already enabled by the agent in order to use the module.

Uses design contract can be seen as the public interface which permits reuse modules code, and will be marked as public\(^4\). First list stands for those components that agent “import” from module; while second list, stands for the components that module needs to “import”, e.g. from agent. Note that both lists are defined at modules side. From this perspective we tackle the following issues:

- **Modular hierarchized agent behaviour design.** Capabilities represented by habilitation list are generally associated to a higher-abstraction level behaviour than those that requirement list represents. Our notion here is based in the fact that in a general sense complex behaviour is compounded by an organized set of simpler behaviours, so it is possible to conceive agent programs design into different levels of abstraction (see figure 1).

![Fig. 1: Modules Herarchization. Since agent already posses GetBalance capability it can load Cyclist Module which enables RideBicycle capability to agent. Next, this is able to fulfill the Rider Module use design contract, and thus acquire the Ride Motocross capability through loading Rider Module, for whose the original source of RideBicycle capability is transparent.](/image)

\(^3\)Internal actions in Jason are seen as atomic components programmed in pure Java which model changes in the agents internal state

\(^4\)Here we use public notion as regular, in the sense of visibility and access, thus a public component within a module is both visible and accessible by the agent which loads the module, while a component not indicated as public remains private, that is visible and accessible only within the module’s scope, in JasMo all components not public are private.
• Avoid name-clashes. All module components are private (except those within the module’s habilitation list), i.e., they are not visible outside the module’s scope. Same way, agent’s components remain hidden to module’s code, except for those in the requirement list. This allows independent development. (see figure 2).

![Fig. 2: Encapsulating plans and Beliefs. Illustrates that modules plans and beliefs are encapsulated into agent’s plan library and agent’s belief base respectively.](image)

• Modules Reusability. By means of the requirements list, seen as a design contract, when writing module’s code there is a way to delegate the concrete implementation of specific capabilities, similarly to interfaces and abstract methods notions in the object-oriented paradigm. This allows to write code with a higher level of abstraction that can be reused by different agents (see figure 3).

![Fig. 3: Use Design Contract. Illustrates how a module can transparently reuse already available capabilities in different agents independently on their environment and agent specific implementation. This promotes modules reusability and independent development.](image)

• Implicit dependency resolution. Dependencies between modules are not established explicitly, this is given transparently when an agent loads one module whose habilitation list fulfils a second module’s requirements list. This way we avoid the need to establish modules dependency relations in the whole system at design time (see figure 1).

• Flexibility and low coupling. Expressing module’s component dependencies in terms of capabilities requirements, leads to a higher flexibility, in the sense that: i) such capabilities can be fulfilled by different agents in distinct ways; and ii) agent only shares a precise list of components to each module according to module’s specific use design contract. This also aims a general lower-coupling between agents and modules than that reached by an opposite approach in where such dependencies are indicated through a single export statement within agent’s code (see figure 4).

![Fig. 4: Low-Coupling. Agent only share and make public specific list of components to each module.](image)

• Autonomy. Specifying at module’s side the uses design contract lists, we also delegate to agent the authority to accept a specific module’s use design contract or not.

As mentioned above we use predicate indicators in the form of a pair functor/arity to reference Jason language components in the use design contract. Language components that can be referenced in the habilitation list are triggering events predicates, each as a capability for dealing with a particular task; and sets of relevant beliefs. On the other hand, components that can be referenced in the requirements list include also actions. Since actions depend directly on environment and agent architecture implementation, when referenced within a module’s code, those are always automatically included into module’s requirements list even if not explicitly written.

In order to well encapsulate plans and beliefs loaded from a module and avoid name-clashes, it is necessary that when an agent loads a module, events generated from components defined within a module’s plan that is encapsulated into an

5All those beliefs which correspond a particular predicate indicator, e.g. for a predicate indicator state/1, we say that state(empty), state(full), state(low) are relevant beliefs.

6Agent architecture in Jason provides perception (which models the agent’s ‘sensors’), acting (modelling the agent’s ‘effectors’), and how the agent receives messages from other agents.
agent must be distinguished from those generated from agent’s “original” plans, and remain only visible within such module scope, that is, for a goal like \( g \) generated within a plan body loaded from a module, only plans loaded from same module must be considered to compute the set of relevant plans. Same way, to compute any logical consequence test inside a module-loaded plan’s body, only same module beliefs must be considered as relevant. To achieve this above, when an agent loads a module it creates a new module scope, and makes reference to this through a local identifier, then each module-loaded component is associated to its corresponding module scope by means of that local identifier. For those components not associated to any module, we say they are in agent’s scope. Then an agent loads each module component dynamically under a corresponding module scope. In fact, it is also possible to load the same module under different scopes, by designating distinct identifiers to each module loading.

Our modularization framework encapsulates components following next guidelines:

- A triggering event generated by an achievement goal addition/deletion, a test goal addition/deletion or a belief base update, within a module scope must only consider plans associated to same module scope to generate the set of relevant/applicable plans.
- A belief addition within a module scope adds a belief associated to the corresponding module scope within the agent belief base.
- Module components listed in habilitation list are never encapsulated into modules. That is, they are not associated to any module scope and remain always visible to agents.
- Agent components listed in the requirements list become visible to module when loaded by any agent fulfilling the module’s use design contract. All other agent components must remain hidden and separated from code within the module.
- When referenced within a module, actions, internal actions and standard internal actions are never encapsulated by modules, i.e. these are always part of the module’s requirements list.
- A triggering event generated by an achievement goal addition/deletion, test goal addition/deletion or a belief base update under an agent assigned local identifier called module scope, is only option to deal with such achievement goal. Note that if module 1 scope, it is considered for events associated to agent scope as well, since predicate do/1 is in module 1 habilitation list (as pointed in our framework guidelines, components in grants list are not encapsulated, so they are not associated to any module scope, which means that they are loaded under the agent scope). Finally, because of belief indicator \( \text{bel(same)} \) is referenced by predicate indicator \( \text{bel/1} \) in module 1 requirements list, and agent effectively beliefs \( \text{bel(same)} \), plan p1 is applicable and executed.

Agent’s belief base reflects the belief addition event generated at line 2 associated to m1 scope. In line 3, an event within module scope triggers plan p2 execution, what causes belief \( \text{bel2(same)} \) being added to the agent’s beliefs set under m1 scope, and the execution of plan with header +/do3 in agent scope. Finally, p2 plan is discarded as relevant by /do2 at line 1, and thus the plan that prints “agent do2” is selected as the only option to deal with such achievement goal. Note that if we omit line 2 the only agent intention formed will fail at line 3 since it will not be applicable plan for +/do2.

**IV. Formal Semantics**

**A. Syntax**

We define a module \( K \) as a tuple \( \langle ds, bs, ps \rangle \) where the component \( K_{bs} \) is a set of annotated ground atomic formul\( \text{a} \) of the form \( \{af_1[s_1, \ldots, s_n], \ldots, af_n[s_1, \ldots, s_n]\} \); \( K_{ps} \) is a set

\[\text{af}_1[s_1, \ldots, s_n], \ldots, \text{af}_n[s_1, \ldots, s_n]\]
\{p_1, \ldots, p_n\} \text{ where each element } p \text{ is a plan. Both components hold the same definition described in [1] described in section 2, so modules are directly reusable code for any AgentSpeak agent. Finally, } K_{ds} \text{ designates the use design contract of the module through a set of predicate indicators with the form } P/a, \text{ where } P \text{ is the predicate functor and } a \text{ is an integer number which represents the predicate arity. }

Here we use public notion as regular, in the sense of visibility and access, thus a public predicate within a module is both visible and accessible by the agent which loads the module, while a predicate not indicated as public remains private, that is visible and accessible only within the module scope. Also we distinguish two types of public elements: i) those that an agent can use from a module, representing the capabilities that a module endows to an agent when loaded, referenced by the set hs; and ii) predicates that the module uses from agent, representing the capabilities that an agent must posses in order to load a module, denoted by the set rs. Both hs and rs are subsets of ds. The complete syntactic description of a module is as follows:

\[
\text{module ::= ds bs ps} \\
ds ::= hs rs \\
hs ::= \text{pr}_1, \ldots, \text{pr}_n \ (n > 0) \\
rs ::= \text{pr}_{n+1}, \ldots, \text{pr}_m \ (m > n) \\
pr ::= P/a \ (a \geq 0)
\]

B. Semantics

First, in order to expose JasMo semantics it is necessary to describe the configuration tuple used to define Jason semantics and point that this is given by a set of rules that define a transition relation\(^8\) over the configuration \(\langle ag, C, M, T, s \rangle\), where:

- \(ag\) is an agent program formed by a set of beliefs bs and a set of plans ps (as defined in the grammar above).
- An agent’s circumstance \(C\) is a tuple \(\langle I, E, A\rangle\) where \(I, E\) and \(A\) are a stack of instantiated intentions, an events queue and a set of actions to be performed.
- \(M\) is the component that register the communication aspects of agents.
- \(T\) keeps a track of temporary information that is required in subsequent stages within a single reasoning cycle. Is conformed by a tuple \(\langle R, Ap, i, \epsilon, \rho \rangle\) where \(R\) is the set of relevant plans for the event being handled, \(Ap\) denoting applicableplans is the set of relevant plans whose contexts are believed true, and \(i, \epsilon, \rho\) record a particular intention, event and applicable plan respectively, being considered along the execution of one reasoning cycle.
- \(s\) indicates the current step within an agent’s reasoning cycle, \(s \in \{ \ldots, \text{ExecInt}, \text{ClrInt} \}\) where last two stand for executing the selected intention and clearing an intention.

Then we say that an atomic formulæ \(at\) is a private element owned by module \(K\) if \(\text{predind}(at) \notin K_{ds}\) where \(\text{predind}\) is a syntactic function which returns the predicate indicator \(pi\) of \(at\). Then in order to support modularization, an agent must differentiate beliefs, plans and events scopes.

For this above, we make use of Jason annotations mechanism in the way that every belief, plan and triggering event associated to a module scope, will be annotated with source(id) where id is the scope identifier of the corresponding module. In order to do so we define the accessory function \(K' = \text{parse}(K, id)\), where the second parameter is an arbitrary atom designated by the agent as a local identifier to the loaded module, and \(K'\) is generated by means of parsing and adding to every private atomic formulæ in \(K\) the source annotation id. So, if an agent loads \(K\) under the local identifier id, our function will parse \(K\) to enrich each \(at \in K\) with the source annotation id, except for those \(at's\) where \(\text{predind}(at) \in ds\). In other words \(\forall at \in K\) with annotations \(S = \{s_1, \ldots, s_n\}\) and where \(\text{predind}(at) \notin ds\), \hspace{1em}\text{we redefine} \hspace{1em}S \text{ as } S \cup \text{source}(id).\) This way, all plan \(p \in K_p\) and every belief \(b \in K_b\) not indicated as public will be annotated with source id. Since every not public at is explicitly annotated, all events generated will be annotated with source id as well, so events are associated to their corresponding scope.

Then, if we say that an agent \(ag\) loads dynamically a module by means of the internal action \text{jasmo.load}(K, id),\ where the first parameter is a module, and the second parameter is an atom as the local identifier for the scope in which \(K\) components will be loaded. The load can be explained by the following semantic rule:

\[
\text{load}(T) = i[\text{head} \leftarrow \text{load}(K, id); b]\ 
\begin{align*}
&\text{loaded}(id) \neq \text{ag} \\
&\langle ag, C, M, T, \text{ExecInt} \rangle \rightarrow \langle ag', C, M, T, \text{ClrInt} \rangle \\
&K' = \text{parse}(K, id) \\
&ag' = ag \cup K' \\
&ag_{bs} = ag_{bs} \cup K_{bs} \\
&ag_{ps} = ag_{ps} \cup K_{ps} \\
&ag_{bs} = ag_{bs} + \text{loaded}(id)
\end{align*}
\]

Now to avoid cluttering the formal semantics, we use \(Q = \{id_1, \ldots, id_n\}\) to denote the set of scopes identifiers loaded by a single agent, then we say that every \(at\) not annotated with a source id such that \(id \notin Q\), is associated to agent’s scope.

Regarding logical consequence operation in Jason [1], note that some like \(\text{bel[source(id)]} = \text{bel}\) is always true\(^9\), but if we want to support encapsulated beliefs as described above, that only should be true if \(id \notin Q\), that is, annotated bel is not associated to any module scope. To reflect so, it is necessary to extend the way in which the logical consequence operator computes the belief base\(^10\).

Then, we can say that only in the case that an atomic formulæ \(at_1\) have annotations \(S_1 = \{\}\), \(at_1\) is a logical consequence of a set of ground atomic formulæ, written \(bs \models at_1\) if, and only if, there exists \(at_2[\{s_2, \ldots, s_n\}] \in bs\) such that (i) \(at_1 \theta = at_2\), for some most general unifier \(\theta\), and (ii) \(S_2 \setminus (S_2 \cap Q) \neq \{\}\); in some other case (e.g. \(S_1 \neq \{\}\)), we

\(^8\)For a complete description of the formal semantics and customizable components refer to [1]

\(^9\)In Jason, beliefs written within the context of a plan and that have not been explicitly annotated, are not assumed to have source(self), this generates a problem since something like bel is modelled by any bel[\{S_1, \ldots, S_n\}], and for the case of JasMo this is not always true, i.e., when some \(S_i = \text{source}(id)\) such that \(id \in Q\).

\(^{10}\)We do not use an explicit source annotation for agent scope to permit Jason agents to be completely compatible with JasMo agents.
apply the regular definition for $\models$ operator described in \cite{1}.

Finally, agents $ag$ can dynamically unload a module. This is described by the next operational semantic rule\footnote{Here $i[p]$ denotes the intention that has plan $p$ on top of intention $i$.}

\[
\text{unload}(T_i = \langle \text{head} \leftarrow .\text{unload}(id); h \rangle \models \text{loaded}(id) \models ag_{\langle ag', C, M, T, \text{ExecInt} \rangle} \rightarrow \langle ag', C, M, T, \text{ClrInt} \rangle)
\]

\[
\begin{align*}
ag'_{bs} &= ag_{bs} \setminus K_i'_{ids} \\
ag'_{ps} &= ag_{ps} \setminus K_i'_{ids} \\
ag_{bs} &= ag_{bs} - \text{loaded}(id)
\end{align*}
\]

V. IMPLEMENTATION

Algorithm 1 implements the $\text{load}$ rule. This basically parses module code in order to be loaded by the agent. Basically we add the module $id$ annotation to each predicate coded within module’s source except for those included in $ds$.

Next we add to agent’s belief base and plan library all initial beliefs and plans defined within the module. Finally, the special belief $\text{loaded}(id)$ is added to denote that the module have been loaded successfully and that $id$ is its module scope identifier.

**Algorithm 1: .load() internal action algorithm for support modules in Jason**

\begin{verbatim}
begin
  Input : Module Source Code File $src$
  Input : Module Identifier $id$
  parser = parse($src$)
  while parser has next do
    token = parser.next()
    if token instance of $at$ then
      if $\text{predind}(token) \notin ds$ then
        do source $id$ to token
      else
        do plan $token$ to agent
      end
    end
    if token instance of $b$ then
      do add believe $loadedM(id)$ to agent
    end
  end
end
\end{verbatim}

Since the extension of the logical consequence operator in Jason is not part of the customizable elements of the language, it is not possible to modify it to make it match our formal definition. To get the same result, we extended the Agent class in Jason and overwrote the $\text{selectOption}$ that implements the $S_O$ function defined in the formal semantics of Jason. We define this function by coding the algorithm 2 which is the general process to get a single plan from the set of applicable plans to conform the intention. In general, for events associated to a module scope, all plans not associated to such scope are discarded.

In a similar way we extended also the $\text{BeliefBase}$ Jason class in order to completely support our notion of logical consequence. We overwrote the method $\text{getCandidateBeliefs}$ which return the relevant beliefs to a certain predicate indicator. In our own version, when a logical consequence is computed into a module scope, all relevant beliefs associated to another scope are simply discarded.

Finally, algorithm 3 describes the general process to unload a module. This removes all plans loaded from a module and all beliefs associated to a module scope.

**Algorithm 2: $S'_O$ function algorithm for support modules in Jason**

\begin{verbatim}
begin
  Input : Set of Applicable Plans $Options$
  Input : Current Triggering Event in Agent’s Circumstance $T_e$
  Output: Selected Plan $p$
  if $\text{source}(T_e) \in O$ then
    foreach $p$ in $Options$ do
      if $\text{source}(p) \notin O$ then
        remove $p$ from $Options$
    end
    return $S_O(Options)$
end
\end{verbatim}

**Algorithm 3: .unload() internal action algorithm for support unload modules in Jason**

\begin{verbatim}
begin
  Input : Module Identifier $id$
  .findall(X[source($id$)], X[source($id$)], $\text{RelBels}_{id}$)
  foreach $\text{bel} \in \text{RelBels}_{id}$ do
    do del believe $\text{bel}$ from agent
  endforeach
  foreach $p \in \text{agent}_{ps}$ do
    if $\text{TrEv}(p)$ hasSource $id$ then
      do drop $p$ from agent
      do del believe $\text{loaded}(id)$ to agent
    end
  endforeach
end
\end{verbatim}

VI. AN EXAMPLE

Here we will use the gold miners agents example. We create a Multi-Agent system to define the agent $\text{Manager}$ as a regular Jason agent and $\text{Worker}$ as a JasMo agent. To do this it is necessary to change both, the defaults agent class and belief base class.

```java
1  MAS modjasn { 2    infrastructure: Centralised 3    environment: minersworld.Minersworld 4    agents: 5      worker agentClass Jason.ModularAgent 6      beliefBaseClass Jasmo.Modularbeliefbase 7    } 8  manager; 9}
```

Fig. 6: MAS Multi-Agent System source code.

Manager agent first will start every worker agent it knows. It has two main plans, one for sending a worker agent to
explore and other for sending a worker to carry the gold at some position. Those plans are executed when a worker notifies that gold has been found at some position, and when a worker agent says it is free.

```java
1. worker(worker2);  
2. worker(worker3);  
3. worker(worker4);  
4. instant.  
5. start.  
6. if start: for (findall w where worker(w,X)) {  
    7. send(w,achieve,star1);  
8. }  
9. -goldAt(POS)[source(X)]: worker(X)  
10. -free[POS]: worker(Y)  
11. -send(Y,achieve,explore).  
```

**Fig. 7:** Manager Agent source code.

The worker agent simply loads explorer and carrier competences and begins to explore.

```java
1. boss(manager).  
2. -instant[source(X)]: boss(X).  
3. -jason.load("explorer,act,c", exc);  
4. -jason.load("carrier,act,c", car);  
5. -explore.  
```

**Fig. 8:** Worker Agent source code.

Explorer module source file starts with two Jason directives denoting the modules’ use design contract. The plan explore do the agent to move itself to some random position first. Note that module explorer does not define a precise way to execute the action moveTo/1, this must be an already capability enabled by the agent. Explorer module requires as well to know who is the boss (boss/1) and how to sense for gold at some position (senseGold/1). Also, it enables agents using this module to know that there is gold at some position (goldAt/1).

```java
1. [enables(“explore”, “goldAt/1”)]  
2. [requires(“boss/1”, “senseGold/1”, “moveTo/1”)];  
3. -explore.  
4. -explore[source(A)]: boss(A).  
5. -getRandomPosition(POS);  
6. -moveTo(POS);  
7. -senseGold(POS);  
8. -freemap(POS).  
9. -positionExplorer(POS).  
```

**Fig. 9:** Explorer module source code.

Finally, the Carrier module allows an agent to fetch and store the gold. In order to do so this module requires to know that there is gold at some position. Note that this last requirement (goldAt/1) will be full filled by the explorer module’s habilitation list.

```java
1. [enables(“carry/1”)]  
2. [requires(“boss/1”, “goldAt/1”)];  
3. -store.  
4. -store[POS][source(A)]: boss(A) & goldAt(POS)  
5. -fetchgold(POS);  
6. -storedGold();  
7. -send(POS,free);  
8. -send(POS,free).  
```

**Fig. 10:** Carrier module source code.

VII. CONCLUSIONS AND FUTURE WORK

On this paper we have argued how a modular agent approach can facilitate the build of extensive agent programs, and proposed and implemented a framework to do so. Moreover we matched our implementation into the formal semantic definition of Jason.

As future work we aim to provide JasMo agents with support for reasoning on ontologies described in OWL. Define the use design contract semantically in order that agents be able to reason about what modules they need to perform a particular task, and solve module dependencies by reasoning on semantic modules descriptions.

Our vision is that JasMo modules can be shared to be stored in public repositories for their use by any agent. And an agent can iteratively get the needed modules to incorporate this specific capability in order to deal with a particular task, this in a similar way as actually a JasMo agent loads a single module, for example:

```
jasmo.getCapability("exploring", id);
```

where exploring represents a capability instead of a particular module.

REFERENCES


