A Namespace Approach for Modularity in BDI Programming Languages

Gustavo Ortiz-Hernández (✉)[1,2], Jomi F. Hübner[3], Rafael H. Bordini[4], Alejandro Guerra-Hernández[2], Guillermo J. Hoyos-Rivera[2], and Nicandro Cruz-Ramírez[2]

[1] Centro de Investigaciones en Inteligencia Artificial - UV
Xalapa, México
[2] Institute Henri Fayol - MINES
Saint-Étienne, France
[3] Federal University of Santa Catarina
Florianópolis, SC, Brazil
[4] FACIN-PUCRS
Porto Alegre, RS, Brazil

Abstract. In this paper we propose a model for designing Belief-Desire-Intention (BDI) agents under the principles of modularity. We aim to encapsulate agent functionalities expressed as BDI abstractions into independent, reusable and easier to maintain units of code, which agents can dynamically load. The general idea of our approach is to exploit the notion of *namespace* to organize components such as beliefs, plans and goals. This approach allowed us to address the name-collision problem, providing interface and information hiding features for modules. Although the proposal is suitable for agent-oriented programming languages in general, we present concrete examples in Jason.

Keywords: Agent-Oriented Programming, Modularity, Namespace

1 Introduction

In the last decades, several programming paradigms have arisen, often presented as an evolution of their predecessors, and with the main purpose of abstracting more complex and larger systems in a more natural and simpler way. Particularly, the Agent-Oriented-Programming (AOP) paradigm has been promoted as a suitable option to deal with the challenges arising when developing modern systems. This paradigm offers high-level abstractions which facilitate the design of large-scale and complex software systems, and also allows software engineers to employ a suite of well-known strategies for dealing with complexity, i.e., decomposition, abstraction and hierarchy.

These strategies are usually applied at the Multi-Agent-System (MAS) level [19,8,15]. However, even a single agent is intrinsically a complex system, hence its design and development should consider the above mentioned strategies. Regarding this, the principle of *modularity* applied to individual agent development can significantly improve and facilitate the construction of agents.

In this paper, we present an approach for programming agents following the principle of modularity, i.e., to develop agent programs into separate, independent, reusable and easier to maintain units of code. In order to support modularity, we identify three major issues to be addressed: i) a mechanism to avoid name-collision, ii) fulfilling the information hiding principle, and iii) providing module interfaces.

Our contribution is to address these issues by simply introducing the notion of *namespace* in the AOP paradigm. In the context of BDI languages, which is the focus of this paper, the novelty of our approach is that it offers a syntactic level solution, independent of the operational semantics of some language in particular, which simplifies its implementation.

The rest of this paper is organized as follows: related and previous work are presented in Section 2; our proposal is described in Section 3; we explain details of implementation in Section 4 and offer an example in Section 5; an evaluation is presented in Section 6; finally, we discuss and conclude in Sections 7 and 8 respectively.

2 Related Work

There exist much work supporting and implementing the idea of modularity in BDI languages. An approach presented by Busetta et al. [6] consists in encapsulating beliefs, plans and goals that functionally belong together in a common scope called capability. The programmer can specify a set of scoping rules to say which elements are accessible to other capabilities. An implementation is developed for JACK [17]. Further, L. Braubach et al. [3] extend the capability concept to fullfill the information hiding principle by ensuring that all elements in a capability are part of exactly one capability and remain hidden from outside, guaranteeing that the information hiding principle is not violated. An implementation for JADEX [4] is provided. Both approaches propose an explicit import/export mechanism for defining the interface.

The modules proposed by Dastani et al. [12] are conceived as separate mental states. This modules are instantiated, updated, executed and tested using a set of predefined operations. Executing a module means that the agent starts reasoning in a particular mental state until a predefined condition holds. This approach is extended by Cap et al. [7], by introducing the notion of sharing scopes to mainly enhance the interface. Shared scopes allow modules posting events, so that these are visible to other modules sharing the same scope. These ideas are conceived in the context of 2APL [10] and an implementation is described in [11].

Also following the notion of capability, Madden and Logan [21] propose a modularity approach based on XML's strategy of namespaces [5], such that each module is considered as a separate and unique namespace identified by an URI. They propose to handle a local belief-base, local goal-base and local events-queue for each module, and then to specify, by means of an export/import statement, which beliefs, goals and events are visible to other modules. In this system, there is only one instance of each module, i.e., references to the exported part of the

module are shared between all other modules that import it. These ideas are supported by the Jason+ language, implemented by Logan and Kiss [9].

Another work tackling the name-collision issue is presented by G. Ortiz et al. [23]. They use annotations to label beliefs, plans and events with a source according to the module to which they belong. In this approach, modules are composed by a set of beliefs, plans and a list of exported and imported elements. Both imported and exported elements are added to a unique common scope. An implementation of this approach is developed as a library that extends Jason.

In Hindriks [16], a notion of module inspired by what they call policy-based intentions is proposed for GOAL. A module is designated with a mental state condition, and when that condition is satisfied, the module becomes the agent focus of execution, temporarily dismissing any other goal. They focus on isolating goals/events to avoid the pursuit of contradictory goals.

In Riemsdijk et al. [26], modules are associated with a specific goal and they are executed only to achieve the goal for which they are intended to be used. In this approach, every goal is dispatched to a specific module. Then all plans in the module are executed one-by-one in the pursuit of such goal until it has been achieved, or every plan has been tried. This proposal is presented in the context of 3APL [13].

A comparative overview of these approaches is given in Table 1. All solutions tackle the name-collision problem, providing a mechanism to scope the visibility of goal/events to a particular set of elements, e.g., plans. They also offer different approaches for providing the interface of modules. However, not all of them fulfill the information hiding principle.

It is also worth mentioning that all those approaches propose some particular operational semantics tied to the AOP language in which they have been conceived and implemented. The proposal that we present in this paper provides a mechanism to address those issues independently of the operational semantics.

3 Modules and Namespaces

A *module* is as a set of beliefs, goals and plans, as a usual agent program, and every agent has one initial module (its initial program) into which other modules can possibly be loaded. We refer to the beliefs, plans and goals within a module as the *module components* (cf. Figure 1).

Modularity is supported through the simple concept of *namespace*, defined as an abstract container created to hold a logical grouping of components. All components can be prefixed with an explicit namespace reference. We write `zoo::color(seal,blue)` to indicate that the belief `color(seal,blue)` is associated with the namespace identified by `zoo`. Furthermore, note that the belief `zoo::color(seal,blue)` is not the same belief as `office::color(seal,blue)` since they are in different namespaces.

Namespaces are either *global* or *local*. A global namespace can be used by any module; more precisely, the components associated with a global namespace

approach	IL	IS	IH	NC	interface's mechanism
Busetta et al. [6]	JACK	✗	✓	✓	explicit import/export
Braubach et al. [3]	JADEX	✗	✓	✓	explicit import/export
Dastani et al. [12]	2APL	✗	✗	✓	set of predefined operations
Cap et al. [7]	2APL	✗	✗	✓	sharing scopes
Madden et al. [21]	Jason+	✗	✓	✓	explicit import/export
Hindriks [16]	GOAL	✗	✗	✓	mental-state condition
Riemsdijk et al. [26]	3APL	✗	✗	✓	goal dispatching
Ortiz et al. [23]	Jason	✗	✗	✓	unique-common scope
Our Proposal	Jason	✓	✓	✓	global namespaces

Table 1: The columns represent existing features in the surveyed approaches, in respect to the issues mentioned in Section 1. The abbreviations stand for: (IL) implementing language; (IS) the approach is independent of the language's operational semantics; (IH) fulfills the information hiding principle; (NC) provides a mechanism to deal with the name-collision issue. The last column refers to the general notion used to provide an interface.

can be consulted and changed by any module. A local namespace can be used only by the module that has defined the namespace.

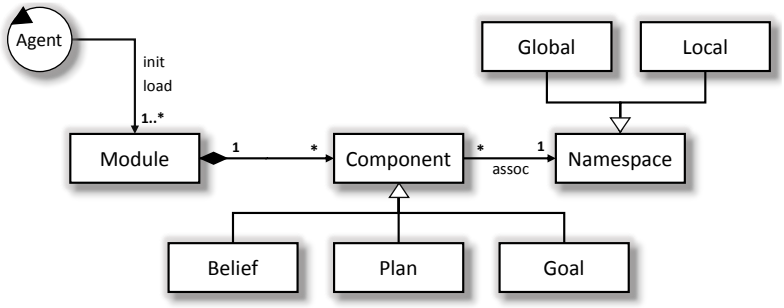

Figure 1: Proposed model for modularity.

We introduce the notion of *abstract namespace* of a module to denote a namespace whose name is undefined at design-time, and will be defined at run-time when the module is loaded. To indicate that a component is in a module's abstract namespace, the prefix is simply omitted, e.g., a belief written as `taste(candy,good)` is in an abstract namespace and its actual namespace will be defined when the module is loaded.

The module loading process involves associating every component in the abstract namespace of the module with a concrete namespace, and then simply incorporating the module components into the agent that loaded the module. Therefore, a concrete namespace must be specified at loading time to replace the module's abstract namespace.

When a module (the loader) loads another module (the loaded), they interact in two directions: the loader *imports* the loaded module components associated with global namespaces and the loader *extends* the functionality of the module by placing components in those namespaces. Figure 2 illustrates the interaction when a module A loads some module B.

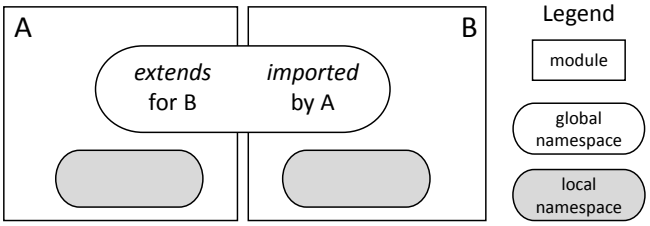

Figure 2: The interaction between modules.

A module is formally defined as a tuple:

$$mod = \langle bs, ps, gs \rangle$$

where $bs = \{b_1, \ldots, b_n\}$ is a set of beliefs, $ps = \{p_1, \ldots, p_n\}$ is a set of plans, and $gs = \{g_1, \ldots, g_n\}$ a set of goals. As shown in Figure 1, each of these components is associated with a namespace. We use subscripts to denote the elements of a module, e.g., mod_{bs} stands for the beliefs included in module mod.

3.1 Syntax

As in many programming languages, we use identifiers to refer to the module components, i.e., its beliefs, plans and goals. Since the syntactic identifiers depend on the programming language and our proposal is intended to be language independent, we propose to extend the syntax of identifiers allowing a namespace prefix:

$$\langle id \rangle ::= [\ \langle nid \rangle ::]\ \langle natid \rangle$$

where *nid* is a namespace identifier and *natid* is used to denote the native identifiers of some AOP language. For example, a belief formula like `count(0)`, whose identifier is `count`, can be written `ns2::count(0)` to associate the belief with namespace `ns2`.

We use a syntactical name-mangling technique[5] to associate every component in the abstract namespace of a particular module to a concrete namespace and to bring support for local namespaces. Restriction access to local namespaces is implemented by replacing every local namespace identifier in the components of a particular module by an internally created identifier. This is generated in such a way that it is not a valid identifier according to the grammar of the native

[5] A technique used in programming languages, which consists in attaching additional information to an identifier, typically used to solve name conflicts.

Algorithm 1: The $mangling(src, nid)$ function associates each component in the abstract namespace of a module program src with a concrete namespace nid and renames local namespaces with an internally generated identifier.

```
1 begin
      Input: src : a module program
      Input: nid : a concrete namespace
2     mod = parse(src)
3     foreach id ∈ ids(mod) do
4         if ns(id) is an abstract namespace then
5             replace id by nid::id in mod
6         if ns(id) is local then
7             replace id by #nid::id in mod
8     return mod
```

language. For instance, if `ns2` is the identifier of a local namespace, the mangling function renames `ns2::color(box,blue)` to `#ns2::color(box,blue)`, where `#ns2` is an invalid identifier and thus no developer can write a program that accesses this belief. We use `#`nid to denote a mapping from nid to an internally generated identifier, unique in the module program where it is being used. The mangling function is described in algorithm 1. To avoid cluttering the notation, we define an auxiliary function $ids(mod) = \{id_1, \ldots id_n\}$ that gets all identifiers id that are in the components bs, ps, and gs of module mod and function $ns(id)$ gives the namespace of identifier id.

3.2 Loading Modules

We represent an agent state as a tuple $ag = \langle B, P, G, \ldots \rangle$, where $B = \{b_1, \ldots, b_n\}$ stands for the agent's belief base, $P = \{p_1, \ldots, p_n\}$ a plan library and $G = \{g_1, \ldots, g_n\}$ the goals of the agent.[6] All identifiers used in the beliefs, plans and goals are prefixed with a proper namespace. The dots symbol $(\ldots)$ is used in the agent tuple to denote the existence of other components proper of the agent's mental state (such as intentions, mail box, etc.) that are not relevant for the purpose of presenting our proposal.

When an agent loads a module, it incorporates the module components, i.e., beliefs, plans and goals, into its own belief base, plan library and goals, respectively. A namespace must be specified at loading time to replace the module's abstract namespace with a concrete namespace. A transition rule (**Load**)

[6] Sometimes when referring to intentional agents, a distinction between desires and intentions is highlighted to focus on the commitment of the agent towards some goal. In the agent state we do not take commitment into consideration; a goal $g \in G$ can be either a desire or an intention. However, a goal $g \in gs$ in some module is considered as an initial goal.

presents the dynamics of loading a module in a particular namespace. The condition (upper part) stands for the action `load(`src`,`nid`)` that takes a module program src and a namespace nid as parameters. This rule executes the mangling function on the module and incorporates the module components into the agent's current state, already associated with a proper namespace identifier.

$$(\textbf{Load})\frac{\texttt{load(}src\texttt{,}nid\texttt{)}}{\langle B, P, G, \ldots\rangle \rightarrow \langle B', P', G', \ldots\rangle}$$

$$\begin{aligned} \text{where:} \quad mod &= mangling(src, nid) \\ B' &= B \cup mod_{bs} \\ P' &= P \cup mod_{ps} \\ G' &= G \cup mod_{gs} \end{aligned}$$

The agent's *initial module* is loaded in what we call the *default namespace*. This is a predefined global namespace whose identifier is `default`. The initial module program determines the initial belief base, plan library and goals of the agent. We use src_0 to denote the initial module program. The next transition rule (**Init**) describes the agent's initialization.

$$(\textbf{Init})\frac{src_0}{\langle B, P, G, \ldots\rangle \rightarrow \langle B', P', G', \ldots\rangle}$$

$$\begin{aligned} \text{where:} \quad mod &= mangling(src_0, \texttt{default}) \\ B' &= mod_{bs} \\ P' &= mod_{ps} \\ G' &= mod_{gs} \end{aligned}$$

4 Implementation

We present the implementation of our proposal in Jason [2], a Java-based interpreter for an extended version of AgentSpeak(L) [24]. An agent program in Jason is defined as a set of initial beliefs bs, a set of initial goals gs and a set of plans ps, where each $b \in bs$ is an atomic grounded formula (initial beliefs may also be represented as Prolog style rules). Every plan $p \in ps$ has the form $te : ctx \leftarrow body$, where te stands for a triggering event defining the event that the plan is useful for handling. A plan is considered relevant for execution when an event which matches its trigger element occurs, and applicable when the condition ctx holds. The element $body$ is a finite sequence of actions, goals and belief updates. Actions in Jason can be external or internal. An external action changes the environment, unlike an internal action which is executed internally to the agent. Jason allows the developer to extend the parsing of source code by implementing user-customized directives.

The basic syntactical construct of a Jason program is the atomic formula, which as in logic programming has the form $p(t_1, \ldots, t_n)$, where p is the functor, and each t_i denotes a term that can be either a number, list, string, variable, or a structure that has the same format of a positive literal. We say then that each p in a Jason program is a Jason identifier. For instance, a plan such as:

```
+!go(home) : forecast(sunny) ← walk_to(0,0).
```

contains the following identifiers: `go`, `home`, `forecast`, `sunny` and `walk_to`.

We have extended the syntax of Jason identifiers to allow a namespace prefix.[7] Since Jason identifiers are used for beliefs and goals, by prefixing them with a namespace these components are *scoped* within a particular namespace.[8] So, a plan written as:

```
+!ns1::go(home) : ns2::forecast(sunny) ← +b.
```

will consider only an achievement-goal addition event `+!go(home)` in namespace `ns1`, and a belief `forecast(sunny)` in namespace `ns2`; beliefs and goals in other namespaces are thus not relevant for this plan. Terms within a literal are not changed when a module is loaded. However, terms can be explicitly prefixed with a namespace. A term prefixed with `::` is in the abstract namespace (e.g. in `forecast(::sunny)` the term sunny is associated with the abstract namespace).

Jason keywords (e.g., `source`, `atomic`, `self`, `tell`, ...), strings and numbers are handled as constants and are not associated with namespaces.

The Jason internal action `.include` and parsing directive `include` were extended with a second parameter to implement the dynamics of loading a module as presented in Section 3.2. The first argument is the file with the module's source code and the second argument the global namespace used to replace the abstract namespace. A parsing directive `namespace/2` is provided to define the type of the namespace (local or global) and as a syntactic sugar to facilitate the namespace association of components, so that the identifiers enclosed by this directive will be associated with the specified namespace.

The beliefs related to perception are placed in the default namespace, and thus also the corresponding events (external events generated from perception). This solution keeps backward compatibility with previous source code, since the initial module is loaded in the default namespace and the previous version of Jason does not have modules other than the initial one.

5 Example

This section illustrates our proposal for modules in more detail showing an implementation of the Contract Net Protocol (CNP) [25]. The modules `initiator` and `participant` (Codes 5 and 6) encapsulate the functionality to start and participate in a CNP, respectively. The multi-agent system is composed of the initiator agents `bob` and `alice`, whose initial module code is presented in codes 1 and 2 respectively; and the participant `company A` and `company B` (Codes 3 and 4). In this implementation, every CNP instance takes place in a different namespace to isolate the beliefs and events of each negotiation.

[7] For the unification algorithm used by Jason, we can simply consider the namespace prefix as being part of the predicate symbol of the literal.

[8] Plans are also scoped within a namespace given that their triggering events are based on beliefs or goals.

Agent `bob` statically loads the module initiator twice (lines 1-2) using the directive `include/2`. This agent starts two CNP's for tasks `build(park)` and `build(bridge)` (initial goals in lines 4-5) in namespaces `hall` and `comm`. Each goal is handled by the module instance loaded in the same namespace where the goal is posted.

```
1 {include("initiator.asl",hall)}
2 {include("initiator.asl",comm)}
3
4 !hall::startCNP(build(park)).
5 !comm::startCNP(build(bridge)).
6
7
8
9
10
11
12
```

Code 1: bob.asl

```
1  !start([fix(tv),fix(pc),fix(oven)]).
2
3  +!start([]).
4  +!start([fix(T)|R])
5   <- .include("initiator.asl",T);
6      .add_plan(
7      {+T::winner(W)<-
8       .print("Winner to fix",T,"is",W)
9      });
10     // sub-goal with new focus
11     !!T::startCNP(fix(T));
12     !start(R).
```

Code 2: alice.asl

Agent `alice` starts multiple CNP's. It uses the internal action `.include/2` for dynamically loading the module `initiator`. It starts one CNP for each task in a given list of tasks (line 5). Agent alice *extends* the functionality provided by the module initiator to print in the console the winner as soon as it is known. Namely, it adds one plan to the same namespace where the module is loaded (lines 6-9).

Agent `company A` participates in all CNPs. It loads the module `participant` in every namespace where it listen that a CNP has started (note that the namespace in line 2 of code 3 is a variable). The agent customizes the module by adding beliefs in the same namespace where the module is loaded (lines 3-4). The module uses these beliefs to decide what tasks can be accepted and how much to bid (cf. lines 6-7 of Code 6).

Agent `company B` plays participant only for CNPs started by agent `bob`, and taking place in namespaces `hall` or `comm`. When a CNP starts under these conditions, it loads the module `participant` in the corresponding namespace. The beliefs in lines 8-9 and the plan added in lines 14-19 extend the functionality of the module by setting the strategy for bidding and accepting tasks. This company only accepts tasks for building and its bid depends on the namespace in which the CNP is being carried on. Directive `namespace/2` in line 1 defines the local namespace `supp`. This namespace encapsulates the beliefs used to estimate the final price of tasks (lines 2-5), so that they are inaccessible to other modules.

The `initiator` module provides functionality to start a CNP. It starts with a forward declaration of the local namespace `priv` in line 1. The namespace of `startCNP` (line 11) is abstract and a concrete namespace is given when the module is loaded (cf. lines 1-2 and 5 of Codes 1 and 2, respectively). Because the namespace given to `startCNP` is global (as defined by the loader), this module is *exporting* the plan `@p1`. The identifiers without an explicit namespace between lines 30 and 55 will be placed in the local namespace `priv`. This is used to encapsulate the module's internal functionality, so that the plans to carry out

```
+N::cnpStarted[source(A)]
 <- .include("participant.asl", N);
    +N::price(_,(3*math.random)+10);
    +N::acceptable(fix(_));
    !N::joinCNP[source(A)].
```

Code 3: company A.asl

```
{begin namespace(supp,local)}
   price(bridge,300).
   price(park,150).
   gain(hall,1.5).
   gain(comm,0.8).
{end}

hall::acceptable(build(_)).
comm::acceptable(build(_)).

+N::cnpStarted[source(bob)]
 : .member(N,[hall,comm])
 <- .include("participant.asl",N);
    .add_plan({
       +?N::price(build(T),P)
          : supp::gain(N,G)
          <- ?supp::price(T,M);
             P=M*(1+G)
    });
    !N::joinCNP[source(bob)].
```

Code 4: company B.asl

contracts and announcements are only accessible from within this module (as illustrated in the line 23). Similarly, the beliefs added to memorize the current state of the CNP and the rule in lines 4-8 are private and will not interfere or clash with any other belief of the agent. However, a loader module can retrieve the current state of the CNP by means of plans `@p2` and `@p3`. Figure 3 illustrates the relation (imports and extends) between the modules `alice` and `initiator` using the same notation of Figure 2.

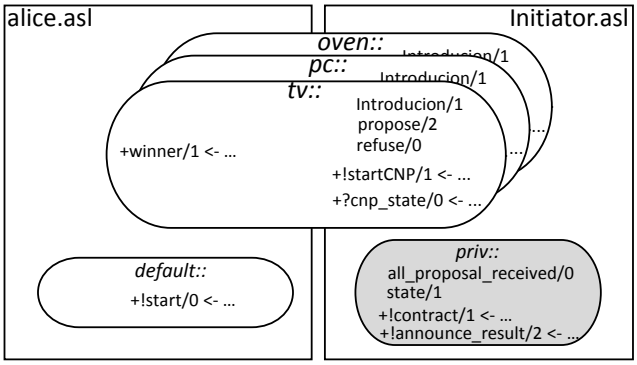

Figure 3: The namespaces of agent alice during its execution.

```
{namespace(priv,local)} //Forward definition

// character :: forces a term to be considered in the abstract namespace
priv::all_proposals_received
 :- .count(::introduction(participant)[source(_)],NP) &
    .count(::propose(_)[source(_)], NO) &
    .count(::refuse[source(_)], NR) &
     NP = NO + NR. // participants = proposals + refusals
```

```
9
10 // starts a CNP
11 @p1 +!startCNP(Task)
12   <- .broadcast(tell, ::cnpStarted); // tell everyone a CNP has started
13      // this_ns is a reference to the namespace where this module is loaded
14      // in this example is the namespace where the CNP is being performed
15      .print("Waiting participants for task ",Task," in ",this_ns,"...");
16      .wait(3000); // wait participants introduction
17      +priv::state(propose); // remember the state of the CNP
18      .findall(A, ::introduction(participant)[source(A)], LP);
19      .print("Sending CFP for ",Task," to ",LP);
20      .send(LP,tell, ::cfp(Task)); //send call for proposals to participants
21      // wait until all proposals are received for a maximum 15secs
22      .wait( priv::all_proposals_received, 15000,_);
23      !priv::contract(this_ns).
24
25 // to let the agent to query the current state of the CNP
26 @p2 +?cnp_state(S) <- ?priv::state(S).
27 @p3 +?cnp_state(none).
28
29 {begin namespace(priv)}
30   //.intend(g) is true if the agent is currently intending !g
31   +!contract(Ns) : state(propose) & not .intend(::contract(_))
32     <- -+state(contract); // updates the state of CNP
33        .findall(offer(Price,A), Ns::propose(Price)[source(A)], L);
34        .print("Offers in CNP taking place in ",Ns," are ",L);
35         L \== [];  // constraint the plan execution to at least one offer
36        .min(L,offer(WOf,WAg)); // sort offers, the first is the best
37        +Ns::winner(WAg);
38        !announce_result(Ns,L);
39        -+state(finished).
40
41   // nothing todo, the current phase is not propose
42   +!contract(_).
43   -!contract(Ns)
44     <- .print("CNP taking place in ",Ns," has failed! None proposals");
45        -+state(finished).
46
47   +!announce_result(_,[]).
48   // announce to the winner
49   +!announce_result(Ns,[offer(_,Ag)|T]) : Ns::winner(Ag)
50     <- .send(Ag,tell, Ns::accept_proposal); // notify the winner
51        !announce_result(Ns,T).
52   // announce to others
53   +!announce_result(Ns,[offer(_,Ag)|T])
54     <- .send(Ag,tell, Ns::reject_proposal);
55        !announce_result(Ns,T).
56 {end}
```

Code 5: initiator.asl

The `participant` module has a plan to join a CNP by sending an introduction message to the agent playing initiator in the corresponding namespace. When a call for proposals is received, an offer is sent back only if the task is supposed to be accepted, otherwise the agent replies with a refuse message (lines 6-13). The accepted tasks and the amount to bid are not provided in the module (lines 6, 7 and 13). They are meant to be defined by a loader module that can extend every instance of this module to specify both tasks to be accepted and the strategy for bidding (e.g. as in modules `company A` and `company B`).

```
// participating in CNP
+!joinCNP[source(A)]
  <- .send(A,tell, ::introduction(participant)).

// Answer to Call For Proposal
+cfp(Task)[source(A)] : acceptable(Task)
  <- ?price(Task,Price);
     .send(A,tell, ::propose(Price));
     +participating(Task).

+cfp(Task)[source(A)]  :  not acceptable(Task)
  <- .send(A,tell, ::refuse);
     .println("Refusing proposal for task ", Task, " from Agent ", A).

// Answer to My Proposal
+accept_proposal : participating(Task)
  <- .print("My proposal in ",this_ns," for task ", Task," won!").
     // do the task and report to initiator
+reject_proposal : participating(Task)
  <- .print("I lost CNP in ",this_ns," for task ",Task,").
```

Code 6: participant.asl

6 Evaluation

We developed a non-modular version of the CNP to compare with the version presented in Section 5. Then, we performed five extensions to both versions. The first consists in modifying the vocabulary used by agents for communication. The second modifies the protocol so that every agent specifies the limit of CNP's in which it is able to participate simultaneously. In the third, initiator agents set a deadline for the call for proposals. The fourth adds one more agent playing initiator and four participants with their own acceptable tasks and strategy to bid. Finally, in the fifth only acceptable proposals are announced.

The comparison among the versions is shown in Table 2. The abbreviations stand for: (num) extension number; (ags) number of agents; (I) number of agents playing initiator; (P) number of agents playing participant; (eds) the number of files edited; (m) modular version, i.e., developed using our approach; (n) non-modular version; (adds) blocks of code added; (dels) blocks of code deleted; (chgs) changes in a line of code. The size of the implementation was calculated after compressing the source files with a zip utility. The initial size is given in bytes, then a percentage representing the increment is given. The extensions are progressive and each is compared against the previous.

For instance, to accomplish extension 2 of the modular version (starting from extension 1), we added six blocks of code and changed two lines across a total of four files, which increased the size of the system programs in 8.2% (i.e., 190 and 195 more bytes than initial implementation and extension 1, respectively) when compared with its previous extension. To extend the corresponding non-modular version, three files were edited to add twelve blocks of code and perform six changes in different lines, increasing the program size in 7% (i.e., 224 and 199 more bytes than initial implementation and extension 1, respectively). The number of agents remained the same in both versions. Total row summarizes the

num	extension	ags		eds		size		updates					
								adds		dels		chgs	
		I	P	m	n	m	n	m	n	m	n	m	n
	initial implementation	2	3	-	-	2359	2864	-	-	-	-	-	-
1	update communication vocabulary	2	3	2	5	-0.5%	0.8%	0	0	0	0	15	37
2	participants set a limit of CNP's	2	3	4	3	8.2%	7.0%	6	12	0	0	2	6
3	initiators set a deadline	2	3	3	2	2.1%	1.5%	3	4	0	0	1	2
4	add more participants	3	7	5	5	50.6%	85.9%	48	126	0	0	0	0
5	participants are not notified if lose	3	7	2	10	-1.3%	-6.6%	0	0	2	10	0	0
	total	-	-	16	25	59.1%	88.6%	57	142	2	10	18	45

Table 2: Comparison of the CNP across a series of extensions.

updates and the increase along all extensions of the system. If the same file had to be edited during two different editions it is counted twice.

The results show that the modular version required a total of 77 updates (57 additions, 2 deletions and 18 changes) against the non-modular for which 197 updates were necessary. In this particular case study we are reducing the maintainability effort by 60% (120 updates less). We can conclude that a project developed using our approach is easier to maintain.

This results can be analyzed in terms of the Don't repeat yourself (DRY) principle.[9] Our proposal enforces this principle since it represents a mechanism to avoid the repetition of code in several parts of the system. In contrast to the non-modular version, where every component implementing the functionality of the protocol is repeated in the program of each agent, the higher the number of participant agents (interested in different tasks and having distinct bidding strategies) the greater the count of repetition occurrences. For instance, if some change is performed in the protocol, even as simple as the way in which participants introduce themselves, the change have to be propagated to the source code of every agent participating in the CNP's.

We made some initial effort in comparing our proposal with the usual `include` directive in previous releases of Jason. Due to the chosen metrics and example, the difference appeared negligible. In future work we will consider other metrics and examples where the difference to a version with the old include directive might be more significant. In any case, it should be emphasized that clearly the old directive does not solve the problem of name collision nor supports information hiding. For instance, if an agent `tom` already uses `price/2` (e.g., to record the prices for supplies), when it includes the source file implementing the CNP (using an include without support for namespaces), since the belief `price/2` is

[9] A principle of software development with the purpose of reducing the repetition of information [18], so that a modification of any single element of a system does not require a change in other logically unrelated elements.

also used by the CNP implementation to determine the bids, a name-collision arises and the resulting behavior is unexpected.[10] For solving this, it is necessary either to change the name of the belief used by `tom` to record the prices of supplies, or that one used in the CNP implementation. Note that the latter alternative implies updating every agent using the source file implementing the CNP.

The following section overviews how this proposal for modules addresses the issues mentioned in Section 1; and highlight some of its properties, as well as the major differences of our approach over related work mentioned in Section 2.

7 Discussion

The notion of namespaces adapted to the context of BDI-AOP languages is suitable to address the main issues related to modularity. For instance, the name-clashing problem is tackled by associating each component to a unique namespace, enabling the programmer to write qualified names for disambiguating references to components.

The interface is provided through the concept of global namespace, which supports both importing components and extending the functionality of modules. The notion of abstract namespace allows dynamic association of module components to namespaces, thus the same module can be loaded several times in different namespaces and also multiple modules can be loaded into the same namespace to compose a more complex solution. The local namespaces permit programmers to encapsulate components which facilitates independent development of modules. Moreover, loading modules at runtime can be seen as dynamic updating, i.e., the acquisition of new capabilities without stopping its execution.

The main difference of our approach resides in the strategy adopted to achieve modularity. On the one hand, the strategy adopted in this paper consists in logically organizing component names in the agent's mental state, by attaching additional information to their identifiers. On the other hand, approaches mentioned in Section 2, in general, are based in mechanisms for dealing with multiple mental states inside the agent, in which modules are active components of the operational semantics of the language, i.e., new transition rules are needed for handling multiple belief bases, plan libraries and/or event queues in the same reasoning cycle. The latter strategy leads to solutions that are more difficult to implement, in contrast to ours, which brings a syntactic level solution, so that it can be implemented in several BDI languages by simply extending their parsers.

7.1 Module relationships

Next, we discuss how our approach is suited to construct association, composition and generalization relationships between modules as defined and analyzed for

[10] This is also reported by N. Madden and B. Logan [21] from the experience of using the usual `include` directive available in previous releases of Jason for the development of a large-scale multi-agent system [20].

capabilities by I. Nunes in [22], as well as some principles from Object-Oriented programming.[11]

Association There exists an association between a *loader* module and a *loaded* module when the execution of at least one plan of the loader requires a goal, whose plan to achieve it, is part of the loaded module. According to [22], association promotes high cohesion by allowing to modularize functionality that addresses different concerns into separate modules.

An example of association is illustrated by Codes 7 and 8. The module `one` loads module `A` and executes one of its plans. The module is loaded using a local namespace to avoid breaking the information hiding principle. In this relationship `A` is not aware of `one`.

```
1 {namespace(ia,local)}
2
3 +!do <- .include("A.asl",ia);
4         !ia::inc(2).
```

Code 7: one.asl

```
1 count(0).
2
3 +!inc(S) : count(X)
4    <- -+count(X+S).
```

Code 8: A.asl

Composition It is a stronger relationship than association. As stated in [22] there exist situations in which the *loaded* module uses components of the *loader* module. This increases the coupling between modules, but allows to model the notion of containment ensuring the information hiding. We missed this type of relationship in our implementation, because we stand for passing information as arguments, when sharing information from *loader* to *loaded* is necessary. In this way, the information hiding principle is not broken and the coupling is reduced.

However, it is possible to model the composition relationship described by I. Nunes using namespaces, by simply adding a symbol to reference the abstract namespace of the loader module.

We provide an example in Codes 9 and 10. The module `B` access to belief `rate/1` in module `two`. The resulting output of executing plan `do/0` in module `two` will be `counter 1`. The symbol ° is used to refer the *loader*'s abstract namespace.[12]

Cardinality Since the same module can be loaded in multiple namespaces while each instance conserves its individual beliefs, it is also possible to represent the cardinality in modules associations. For example, at Codes 2 and 5 in Section 5, module `bob` loads one instance of `initiator` for each contract net protocol it starts, so that each negotiation maintains its own state.

[11] The concept of capability and modules are quite similar, since both are composed of a set of beliefs, plans and goals [6,3], the relationships identified for capabilities can be applied to our notion of modules as well.

[12] This can be supported by extending the mangling function (c.f. algorithm 1) in order to replace the ° symbol by the corresponding namespace at loading time.

```
1 {namespace(ib,local)}
2
3 rate(0.50).
4
5 +!do <- .include("B.asl",ib);
6         !ib::inc(2);
7         ?ib::count(X);
8         .print("counter ",X).
```

Code 9: two.asl

```
1 count(0).
2
3 +!inc(S) : count(X)
4    <- ?°::rate(R);
5       -+count(X+S*R).
6
7
8
```

Code 10: B.asl

Visibility Local namespaces can be used to keep components private within a module, and global namespaces to share components between all modules. However, sometimes it results useful to share components only among the instances of the same module, e.g., in order to avoid replicating the same information several times. It is possible to model this, by introducing a new level of visibility for namespaces (besides *global* and *local*). For instance, a *module* namespace will be accessible only from within all instances of the same module. Comparably to the class visibility level from the Object-Oriented programming as implemented by the modifier `static` in Java.

Multi-inheritance This can be modeled as the union of modules. Typically this union is meant to form a new module which encapsulates a more complex and specialized behavior, while reusing beliefs, plans and goals from other modules.

In the following example (Codes 11 and 12), the module `C` inherits `B` by including all its components in the same namespace (line 9). The *parent*'s local namespaces of each module (if exist) still hidden from the *child* module, and vice-versa. The inclusion of the parent module is performed at the end of the source code, in order to *override* the already existing plan `inc/1` in `A`. This latter works in the particular case of Jason because, by default, the first plan listed in the code is selected for execution, in the case that multiple applicable plans to achieve the same goal exist. A more sophisticated solution for AgentSpeak(L)-style languages is presented by A. Dhaon and R. Collier in [14]. Their method consists in customizing the selection function used for the interpreter to select the next plan for execution, and thus disambiguate which plan must be executed when the same plan is implemented in different levels of the module's hierarchy.

Dynamic extension of modules can be performed too by using the notion of namespaces, in Code 2 at lines 6-9, the functionality of module `initiator` (c.f. code 5) is extended. This is useful in the case that the programmer desires to extend the functionality for only one instance without creating a new module. A similar mechanism is used in Java through the concept of anonymous classes.

A related approach for constructing inheritance relationships between modules in the realm of logic programming, that can be adapted as a more general solution for modeling multi-inheritance in Agent-Oriented programming, is presented by M. Baldoni et al. in [1]. They use a modal operator instead of namespaces to group rules, then a set of logic implications establishes the inheritance

relationship between modules. However, it should be carefully analyzed in order to evaluate its feasibility before adopting it in the context of Agent-Oriented programming.

```
{namespace(ic,local)}

+!init
   <- .include("C.asl",ic);
      !ic::inc(2);
      !ic::mult(2);
      ?ic::count(X);
      .print("counter " X).

```

Code 11: three.asl

```
//belief count/1 is inherited from A
+!mult(T) : count(X)
   <- -+count(X*T).

//overrides plan inc/1 in A
+!inc(S) : count(X)
   <- -+count(X+1).

{include("A.asl")}
```

Code 12: C.asl

8 Conclusion

In this paper we have presented a solution for programming BDI Agents under the principles of modularity, and we explored the assumption that the notion of namespace is enough to address the main issues related to modularity, such as avoiding name-collisions, following the information hiding principle and providing an interface. We have exemplified the properties and feasibility of the approach using the Jason language.

It is future work to provide an unload mechanism that removes components from modules that are no longer used by the agent. We also aim to implement the approach in other languages to further evaluate the generality of the approach.

References

1. Matteo Baldoni, Laura Giordano, and Alberto Martelli. A modal extension of logic programming: Modularity, beliefs and hypothetical reasoning. 1995.
2. Rafael H. Bordini, Jomi F. Hübner, and Michael J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason.* John Wiley & Sons, Ltd, 2007.
3. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *Proceedings of the Third international conference on Programming Multi-Agent Systems*, ProMAS'05, pages 139–155, Berlin, Heidelberg, 2006. Springer-Verlag.
4. Lars Braubach, Er Pokahr, and Winfried Lamersdorf. Jadex: A BDI agent system combining middleware and reasoning. In *Ch. of Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhaeuser, 2005.
5. Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0. W3C recommendation, W3C, August 2006. Published online on August 16th, 2006 at `http://www.w3.org/TR/2006/REC-xml-names-20060816`.
6. Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. Structuring BDI agents in functional clusters. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL 99, Orlando, Florida, USA,*

July 15-17, 1999, Proceedings, volume 1757 of *Lecture Notes in Computer Science*, pages 277–289. Springer, 1999.

7. Michal Cap, Mehdi Dastani, and Maaike Harbers. Belief/goal sharing BDI modules. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '11, pages 1201–1202, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.
8. Pedro Cuesta, Alma Gomez, and JuanCarlos Gonzalez. Agent oriented software engineering. In Antonio Moreno and Juan Pavon, editors, *Issues in Multi-Agent Systems*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 1–31. Birkhäuser Basel, 2008.
9. Bryan Logan Daniel Nicholas Kiss. Jason+ – extension of the jason agent programming language. Technical report, School of Computer Science and Information Technology, University of Nottingham, 2010.
10. Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.
11. Mehdi Dastani, Christian P. Mol, and Bas R. Steunebrink. Modularity in agent programming languages an illustration in extended 2APL. In *In the proceedings of The 11th Pacific Rim International Conference on Multi-Agents (PRIMA), Springer, 2009*, pages 139–152. LNCS, 2009.
12. Mehdi Dastani and Bas Steunebrink. Modularity in BDI-based multi-agent programming languages. In *Proc. of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 02*, WI-IAT '09, pages 581–584, Washington, USA, 2009. IEEE Computer Society.
13. Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A programming language for cognitive agents: Goal directed 3APL. In Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3067 of *LNCS*, pages 111–130. Springer, 2004. 1st International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003.
14. Akshat Dhaon and Rem W. Collier. Multiple inheritance in agentspeak(l)-style programming languages. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 109–120, New York, NY, USA, 2014. ACM.
15. Marie-Pierre Gleizes Federico Bergenti and Franco Zambonelli. Methodologies and software engineering for agent systems. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 4–10. Springer, 2004.
16. Koen Hindriks. Modules as policy-based intentions: modular agent programming in GOAL. In *Proc. of the 5th international conference on Programming multi-agent systems*, ProMAS'07, pages 156–171, Berlin, Heidelberg, 2008. Springer-Verlag.
17. N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - summary of an agent infrastructure. In *Proceedings of the 5th ACM International Conference on Autonomous Agents*, 2001.
18. Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
19. Nicholas R. Jennings. Agent-oriented software engineering. In Ibrahim Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali, editors, *Multiple Approaches to Intelligent Systems*, volume 1611 of *Lecture Notes in Computer Science*, pages 4–10. Springer Berlin Heidelberg, 1999.

20. N. Madden and B. Logan. Collaborative narrative generation in persistent virtual environments. In *Intelligent Narrative Technologies: Papers from the 2007 AAAI Fall Symposium*, Menlo Park, CA, November 2007. AAAI Press.
21. Neil Madden and Brian Logan. Modularity and compositionality in Jason. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers*, volume LNAI 5919, pages 237–253, Budapest, Hungary, 2010. Springer, Springer.
22. Ingrid Nunes. *Improving the Design and Modularity of BDI Agents with Capability Relationships*, pages 58–80. Springer International Publishing, 2014.
23. Gustavo Ortiz-Hernandez, Alejandro Guerra-Hernandez, and Guillermo J. Hoyos-Rivera. JasMo - a modularization framework for Jason. In *12th Mexican International Conference on Artificial Inteligence (MICAI). Mexico City*. IEEE, November 2013.
24. Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
25. R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comp.*, 29(12):1104–1113, 1980.
26. M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-oriented modularity in agent programming. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1271–1278, New York, NY, USA, 2006. ACM.