

Modules and Namespaces in Jason¹

Gustavo Ortiz-Hernández, Jomi F. Hübner, and Rafael H. Bordini

July 2016

[Introduction](#)

[Concepts](#)

[Implementation](#)

[Namespace Prefix](#)

[Loading](#)

[Environment interaction](#)

[Operations](#)

[Unification](#)

[Examples](#)

[Factorial](#)

[Contract Net Protocol](#)

[Properties](#)

[Open Issues](#)

[Bibliography](#)

Introduction

This document describes how modules are used in Jason. Modules in Jason allow the programmer to develop agent programs into separate, independent, reusable and easier to maintain units of code. The introduction of the notion of *namespace* to organize components such as beliefs and events has addressed the name-collision problem providing interface and information hiding features for modules.

Below, we present the conceptual view of modules and namespaces that was implemented for Jason. Examples are used to illustrate the use of these features. At the end of the document, we include some bibliographical references for those interested in alternative ways of having modules in agent-oriented programming languages (some of which have inspired the Jason implementation to some extent).

Concepts

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 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 *namespaces*, 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 `ns1::color(box, blue)` to indicate that the belief `color(box, blue)` is associated with the namespace identified by `ns1`. Furthermore, note that the belief `zoo::color(seal, blue)` is not the same belief as `office::color(seal, blue)`, as they reside in different namespaces.

¹ Another view (more academic) of modules and namespaces for Jason was presented at EMAS@AAMAS 2016.

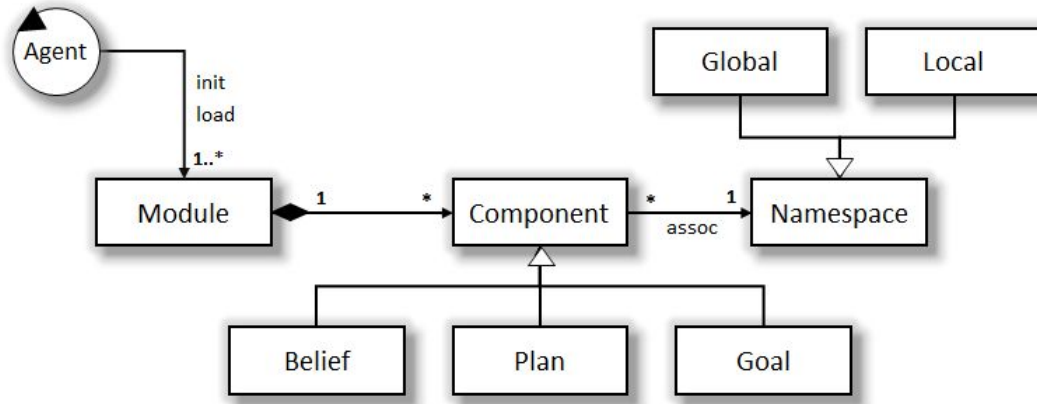


Figure 1. Proposed model for modularity.

Namespaces are either *global* or *local*. A global namespace can be used by any module; more precisely, the components associated with a global namespace can be consulted and changed by any module program. A local namespace can be used only by the module that has defined the namespace. Modules can share components by means of a common global namespace.

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 runtime 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 loader of a module interacts with it in two directions: the loader *imports* the components of the module that are in global namespaces and the loader *extends* the modules by placing components in those namespaces.

Implementation

The basic syntactical construct of a Jason program is a literal, which as in logic programming has the form $p(t_1, \dots, t_n)$, where p is the predicate (that can be strongly negated with the \sim operator), $n \geq 0$, 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 predicate p and structure term in a Jason program is a *Jason identifier*. For instance, in a plan such as:

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

the Jason identifiers are: `go`, `home`, `forecast`, `sunny`, and `walk_to`.

Namespace Prefix

We have extended the syntax of the Jason identifiers to allow a namespace prefix:

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

where *nid* is a namespace identifier (an atom) and *jid* is used to denote the usual Jason identifier. For example, a belief formula like `count(0)` can be written `ns2::count(0)` to associate it with the namespace `ns2`.

Since Jason identifiers are used for beliefs and goals, by prefixing them with a namespace, these elements are being *scoped* within a particular namespace.² Therefore, a plan written as:

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

will consider only an achievement-goal addition event `+!go(home)` in the namespace `ns1`, and a belief `forecast(sunny)` in namespace `ns2`; beliefs and goals in other namespaces are not relevant for this plan.

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

Loading

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

When a module is loaded, its components, i.e., beliefs, plans and goals, are added into the belief base, plan library and desires (i.e. added as goals) of the agent, respectively.

The agent initial module is loaded in what we call the *default* namespace. This is a predefined global namespace whose identifier is `default`. The components in the initial module are used as the initial belief base, plan library, and goals for the agent.

Restricting access to local namespaces is done at loading time, using a *name-mangling* technique. This consists in replacing every reference to a local namespace by an internally created namespace identifier. This is generated in such a way that it is not a valid identifier in the Jason syntax (i.e., programmers cannot access such namespaces). For instance, if `ns2` denotes a local namespace, the loading process will rename a belief `ns2::color(box,blue)` to `#ns2::color(box,blue)` where `#ns2` is not a valid Jason identifier, thus no developer can write a program that accesses this belief.

Terms within a literal are not changed when a module is loaded. For instance, when loading the belief `color(box,blue)` in the namespace `ns2`, the belief `ns2::color(box,blue)` is actually added (and not `ns2::color(ns2::box, ns2::blue)`). Terms can however be used with the namespace prefix, as in `.findall(C, ns2::color(_,C), L)` that will consider only `color/2` beliefs in namespace `ns2`. Briefly, while predicates within module components (goals, belief and plans) without a namespace prefix are in the *abstract* namespace, terms given as argument to predicates are in the *default* namespace. If we need to force some term to be considered in the abstract namespace, we can prefix it with `":"`. For instance, `.findall(C, ::color(_,C), L)` will consider only `color/2` beliefs in the namespace informed when its module was loaded.

Environment interaction

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

² Plans are also scoped within a namespace given that their triggering events are based on beliefs and goals.

code, since the initial module is loaded in this default namespace (besides, it makes sense as all modules of an agent may potentially need to have access to information about the state of the environment).

Operations

The following directives are available to support the use of modules and namespaces in Jason:

- `{ include(<module> [, <namespace>]) }`: loads the module <module> (an .asl file) and its abstract namespace is associated to <namespace>. If the second argument is omitted, the abstract namespace for <module> is the namespace of the module performing the include (and both modules will thus share the namespace). For example, `{ include("m.asl", ns2) }` loads the module "m.asl" using ns2 for its abstract namespace, while if the second argument had not been given, the effect would be the same as the original Jason "include" directive whereby further AgentSpeak code is simply included as if it was part of the file using the include directive.
- `{ namespace(<id> [, <type>]) }`: this directive indicates the type of namespace <id>; the values for <type> can be either `local` or `global`.
- `{ begin namespace(<id> [, <type>]) } ... { end }`: this directive extends the previous directive providing syntactic 'sugar' to facilitate the namespace association of components, so that identifiers in the ... part are placed in namespace <id>.

And the following internal actions:

- `.include(<module> [, <namespace>])`: identical to the its homonymous directive.
- `.namespace(<atom>)`: succeeds if <atom> is a namespace identifier. It backtracks on all global namespaces. For example, `.findall(X, .namespace(NS) & NS::b(X, _::_), L)` put in the list L all first terms of beliefs b/2 from all global namespaces, regardless of the second term in belief b.

Note that the include directive is executed at parsing time (i.e. *static* loading) while the include internal action is executed (from an intention) at run time (i.e. *dynamic* loading).

The operator `=..` now supports four items in the list to include the namespace:

```
bob(10, "ola") [k::annot] =.. [NS, Functor, Terms, Annots]
    NS -> default
    Functor -> bob
    Terms -> [10, "ola"]
    Annots -> [k::annot]
```

Unification

Namespaces are taken into consideration in the unification process, for instance

```
ns1::bel(10) = ns2::bel(10)
```

fails (i.e. it does not unify, as the namespace is treated as part of the predicate name) since we are trying to unify literals in different namespaces. Although we used the unification operator `=` in this example (and below), it could just as well be the case that the agent has `ns1::bel(10)` in its belief base and performed the query `?ns2::bel(10)`. Similarly, although the examples are based on literals, the same applies to

events. The event `!n::g` will have as relevant plans, for instance, plans with the triggering event `!n::g` or `!X::g`.

More examples:

```
bel(10)      = bel(10)          // unifies since both are in the same namespace
n::bel(10)   = n::bel(10)       // unifies since both are in the same namespace
bel(10)      = n::bel(10)       // does not unify
```

Since variables can also be in a namespace, they allow us to further "play" with unification:

```
ns::A = ns::bel(10)           // unifies and A -> bel(10)
ns::A = de::bel(10)           // does not unify
ns::A = bel(10)               // does not unify (unless the abstract namespace is ns)
A      = bel(10)               // unifies and A -> bel(10)
ns::A = ns::B                 // unifies and A -> B
```

Variables can be used as the namespace prefix:

```
ns::A = B::bel(10)            // unifies and A -> bel(10), B -> ns
ns::A = B::C                  // unifies and A -> C, B -> ns
A      = B::C                  // unifies and A -> C, B -> the abstract namespace of A
```

Numbers and strings are in all namespaces by definition (we cannot define their namespaces):

```
A      = 10                    // unifies and A -> 10
ns::A   = 10                    // unifies and A -> 10
N::A    = 10                    // unifies and A -> 10, N -> default
```

It is important to notice that terms also have namespaces:

```
ns::b(k::a) = b(a)             // fails
ns::b(k::a) = ns::b(a)         // fails (the term a is not in the same namespace)
ns::b(k::a) = ns::b(k::a)      // unifies
ns::b(k::a) = N::b(O::a)       // unifies and N -> ns, O -> k
ns::b(k::a) = N::b(O::X)       // unifies and N -> ns, O -> k, X -> a
ns::b(k::a) = N::b(X)          // fails, k::a does not unify with X, different namespaces
```

Examples

In this section, we illustrate how to use the features mentioned above by means of two examples. The first example simply shows how to load a module and execute a plan in it. The second example demonstrates how multiple instances of the same module can be exploited and also how different modules interact.

Factorial

In the following code, the module `initial.asl` uses the internal action `.include` to load the module `factorial.asl` in the namespace `fac` and then adds two subgoals in this namespace. Since these subgoals are posted in the `fac` namespace, they are handled by the module `factorial`.

The module `factorial.asl` provides functionality to print the factorial of a given number. This module defines the *local* namespace `priv` (line 2) to encapsulate the functionality for computing the factorial (lines 3-8), so the beliefs it adds to memoize factorials and the plan to compute them are only accessible from within this module (as illustrated in line 13) and will not interfere or clash with any other module's beliefs or plans. The namespace of `print_factorial` (line 12) is abstract and a concrete namespace is given when the module is loaded. Because the namespace of `print_factorial` is global (as defined by the loader), we say that this module is *exporting* plan @p1.

<pre>1 !start. 2 3 +!start 4 <- .include("factorial.asl", fac); 5 !fac::print_factorial(7); 6 !fac::print_factorial(5). 7 8 9 10 11 12 13 14</pre>	<pre>1 // exports +!print_factorial/1 2 {begin namespace(priv, local)} 3 factorial(0 ,1). 4 5 +?factorial(N,F) : N > 0 6 <- ?factorial (N -1, F1); 7 F = F1 * N; 8 +factorial(N,F) . 9 {end} 10 11 @p1 12 +!print_factorial(N) 13 <- ?priv::factorial(N,F); 14 .print("Factorial of ",N," is ",F) .</pre>
<code>initial.asl</code>	<code>factorial.asl</code>

Contract Net Protocol

This example shows how to implement the Contract Net Protocol (CNP) using modules. Agent Bob (see code `bob.asl`) statically loads the module `initiator.asl` twice (lines 1-2), which endows it to start CNP instances for tasks `build(park)` and `build(bridge)` (lines 4-5). In this implementation, each CNP takes place in a different namespace to isolate the beliefs and events of each independent task allocation process.

<pre>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 13 14 15</pre>	<pre>1 !start([fix(tv), fix(computer), fix(fridge)]) . 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 !!T::startCNP(fix(T)); 11 !start(R) . 12 13 14 15</pre>
--	--

bob.asl**alice.asl**

Agent Alice starts multiple CNP's. Therefore it dynamically loads one instance of module `initiator.asl` for each CNP started (line 5). The functionality provided by the module `initiator` is *extended* by adding one plan to the same namespace where the module is loaded (lines 6-9).

Company A participates in all CNPs by loading module `participant` in every namespace where it listen that a CNP has started (note that the namespace in line 2 is a variable). Two beliefs are added into the namespace where the module is loaded (lines 4-5), extending the module. The module uses these beliefs to decide what tasks can be accepted and how much to bid (cf. lines 6-7 of `participant.asl`).

Company B plays the participant role only in 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.asl` in the corresponding namespace. The beliefs on lines 1-5 extend the functionality of the module by setting the strategy for bidding and accepting tasks. Company B only accepts tasks for building and its bids depend on the namespace in which the CNP is being carried on.

Further details are provided by comments in the sources below.

<pre> 1 2 +N::cnpStarted[source(A)] 3 <- .include("participant.asl", N); 4 +N::price(_, (33*math.random)+100); 5 +N::acceptable(fix(_)); 6 !N::joinCNP[source(A)]. 7 8 9 10 </pre>	<pre> 1 hall::price(build(_),300). 2 hall::acceptable(build(_)). 3 4 comm::price(build(_),100). 5 comm::acceptable(build(_)). 6 7 +N::cnpStarted[source(bob)] 8 : .member(N,[hall,comm]) 9 <- .include("participant.asl", N); 10 !N::joinCNP[source(bob)]. </pre>
---	--

company_A.asl**company_B.asl**

Module `initiator` encapsulates the agent functionality to start a CNP. Since local namespaces have to be defined before their use, a forward declaration of the local namespace `priv` takes place at line 1. The rule on lines 3-7 is `private` because it is added into a local namespace.

```

1 {namespace(priv,local)} //Forward definition
2
3 priv::all_proposals_received
4   :- .count(::introduction(participant)[source(_)],NP) & // number of participants
5     .count(::propose(_)[source(_)], NO) & // number of proposals received
6     .count(::refuse[source(_)], NR) & // number of refusals received
7     NP = NO + NR.
8
9 //Starts a CNP
10 +!startCNP(Task)
11   <- .broadcast(tell, ::cnpStarted);
12     // 'this_ns' is a reference to the namespace where this module was loaded
13     // in this example it is the namespace where the CNP is being performed
14     .print(" Waiting participants for task ",Task," in ",this_ns,"...");
15     .wait(3000);

```

```

16     -+priv::state(propose);
17     .findall(A, ::introduction(participant)[source(A)],LP);
18     .print("Sending CFP for ",Task," to ",LP);
19     .send(LP,tell, ::cfp(Task));
20     // the deadline of the CNP is now +15 seconds, so
21     // the event +!contract(this_ns) is generated at that time
22     .at("now +25 seconds", { +!priv::contract(this_ns) }).
23
24 // if all proposals have been received, don't wait for the deadline
25 // receive proposals
26 +propose(_) : priv::state(propose) & priv::all_proposals_received
27   <- !priv::contract(this_ns).
28
29 // receive refusals
30 +refuse : priv::state(propose) & priv::all_proposals_received
31   <- !priv::contract(this_ns).
32
33 // to let the agent know the current state of the CNP
34 +?cnp_state(S) <- ?priv::state(S).
35 +?cnp_state(none).
36
37 {begin namespace(priv)}
38   +!contract(Ns) : state(propose) & not .intend(::contract(_))
39     <- -+state(contract);
40     .findall(offer(Price,A), Ns::propose(Price)[source(A)],L);
41     .print("Offers in CNP taking place in ",Ns," are ",L);
42     L \== [];
43     .min(L,offer(WOf,WAg));
44     +Ns::winner(WAg);
45     !announce_result(Ns,L);
46     -+state(finished).
47
48
49 // nothing to do, the current phase is not 'propose'
50 +!contract(_).
51
52 -!contract(Ns)
53   <- .print("CNP taking place in ",Ns," has failed! (no proposals)").
54
55 +!announce_result(_,[]).
56 // award contract to the winner
57 +!announce_result(Ns,[offer(_,Ag)|T]) : Ns::winner(Ag)
58   <- .send(Ag,tell, Ns::accept_proposal);
59     !announce_result(Ns,T).
60 // announce to others
61 +!announce_result(Ns,[offer(_,Ag)|T])
62   <- .send(Ag,tell, Ns::reject_proposal);
63     !announce_result(Ns,T).
64
65 {end}

```

initiator.asl

```

1 // Participating in CNP
2 +!joinCNP[source(A)]
3   <- .send(A,tell, ::introduction(participant)).
4

```



```

5 // Answer a Call For Proposal
6 +cfp(Task)[source(A)] : acceptable(Task)
7   <- ?price(Task,Price);
8     .send(A,tell, ::propose(Price));
9     +participating(Task).
10
11 +cfp(Task)[source(A)] : not acceptable(Task)
12   <- .send(A,tell, ::refuse);
13     .println("Refusing proposal for task ", Task, " from Agent ", A).
14
15 // Possibly results of my Proposal
16 +accept_proposal : participating(Task)
17   <- .print("My proposal in ",this_ns," for task ", Task," won!").
18     // do the task and report to initiator
19
20 +reject_proposal : participating(Task)
21   <- .print("I lost CNP in ",this_ns," for task ",Task,".").

```

participant.asl

Properties

Namespaces and modules in Jason have the following properties:

- syntax-based solution (no changes in the semantics of Jason)
- flat scheme for namespaces (no hierarchy)
- the same module can be loaded several times in different namespaces
- naming clash solved by namespaces
- isolation solved by local namespaces
- information hiding solved by local namespaces
- modules can be dynamically loaded (by the `.include` internal action) or statically loaded (by the `include` directive)
- the agent has a single "mind" (avoiding the schizophrenia of the sub-agent approach)
- composition: several modules can be loaded in the same namespace to compose a more complex solution
- event consumption: external events should be consumed by plans in the default namespace and once consumed by a plan cannot be used by others (but a default namespace plan can be used to capture the event and force copies of the event to be recreated for various other namespaces if needed); a side effect: if two modules have plans for the same external event, one will get it and the other no, so the order of loading is relevant in this case.
- the functionality of modules can be extended and customized by adding plans and beliefs to them.

Open Problems

What is the best implementation for `.exclude`/`.unload`?

options (not exclusive):

1. it removes all plans from a namespace (say *n*) -- no reference to the source code file
2. also the local namespaces created in the scope of *n*
3. also beliefs in *n*
4. also intentions from *n*
5. recursively remove also all namespaces used to load some module in the scope of *n*

6. modules have a predefined plan to clean it up (and by default does what options 1-4 suggest). This plan can be called by the loader and can be overridden (as we do with KQML plans).

Bibliography

Paolo Busetta, Nicholas Howden, Ralph Ronnquist, and Andrew Hodgson. *Structuring BDI agents in functional clusters*. In Nicholas R. Jennings and Yves Lesperance, 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.

Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. *Extending the capability concept for exible 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.

Mehdi Dastani and Bas Steunebrink. *Modularity in bdi-based multi-agent programming languages*. In Proceedings 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, DC, USA, 2009. IEEE Computer Society.

Michal Cap, Mehdi Dastani, and Maaïke 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.

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.

Koen Hindriks. *Modules as policy-based intentions: modular agent programming in goal*. In Proceedings of the 5th international conference on Programming multi-agent systems, ProMAS'07, pages 156-171, Berlin, Heidelberg, 2008. Springer-Verlag.

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.

Daniel N. Kiss and Bryan Logan. *Jason+: Extension of the Jason agent programming language*. Dissertation submitted 6th May 2016. School of Computer Science and Information Technology, University of Nottingham, England.