

Agents with Beliefs and Intentions in Netlogo

Ilias Sakellariou

March 2004, Updated March 2008; June 2010

The present document contains two parts, the first describing belief management and the second describing the facilities that allow building proactive (intentions-driven) agents in NetLogo. Thus the combined use of both allows to model and test complex agent architectures in the Netlogo simulation platform.

The code (library) for both belief and intention handling is included in the file *bdi.nls*. NetLogo Versions 4.0 and above allow to have code residing in multiple files and the use of the standard include facilities, through the appropriate `__includes` command (see NetLogo manual for details).

The library is built using the standard programming language provided by NetLogo. The only requirements are that:

- ALL complex agents MUST have two declared -own variables: "**beliefs**" "**intentions**". These are the variables to which beliefs and intentions are recorded. So, in your model if there is a breed of turtles which you wish to model as "BDI" agents, then you should have a **BREED-own [beliefs intentions]** declaration (along with any other variables that you wish to include in your model). MAKE SURE that when you create the variables you initially set their values to empty list (`[]`).
- You also must have **ticks** (see NetLogo manual) in your model or timeout facilities described below will not function.
- Your model should include a "switch" named "**show-intentions**". This is necessary since the code of the receive message checks in each cycle whether to output the messages or not.

Please report bugs to iliass@uom.gr

Have fun with NetLogo.

An Attempt for Belief Management in NetLogo

This paragraph describes an initial version of belief facilities for the NetLogo platform. This primitive library contains a set of necessary procedures and reporters for belief creation and management.

A belief in the library is actually a *list* of two elements: the *type* and the *content*.

- The **belief type** declares the type of the belief, i.e. indicates a “class” that the belief belongs to. Examples could include any string, e.g. “position” “agent” etc. Types facilitate belief management.
- The **belief content** is the content of the certain type of belief. It can be any Netlogo structure (integer, string, list, etc). Notice that there might be multiple beliefs of the same type with a different content, however two beliefs of the same type and content cannot be added.

For example:

[`"agent" 5`] and [`"location" [3 7]`] are examples of beliefs that the agent can have. All agent beliefs are stored in an agent variable *beliefs* that the agent must have (`<breed>-own`) primitive and initially it must be set to the empty list (`set beliefs []`). For example at any given time the beliefs of the agents as stored in that variable can be:

```
[ ["agent" 5] ["location" [3 7]] ["agent" 3] ]
```

Descriptions of the available procedures and reporters for handling beliefs can be found in the sections that follow.

Procedures and Reporters

`create-belief [b-type content]`
(reporter)

The reporter creates a new belief with type *b-type* and *content*, however it **does not store** it in belief memory, but ONLY returns a valid belief. For example:

```
create-belief "agent" 5  
will report a belief ["agent" 5]
```

`belief-type [bel]`
(reporter)

Reports the type of the belief *bel*.

Example: `belief-type ["agent" 5] = "agent"`

belief-content [bel]
(reporter)

Reports the content of belief *bel*

Example: `belief-content ["agent" 5] = 5`

add-belief [bel]
(procedure)

Adds a belief to the beliefs structure. For example:

```
add-belief create-belief "agent" 5
```

will include belief `["agent" 5]` in the beliefs. Multiple such procedures can be invoked. For instance assuming that the beliefs are *initially empty* the following procedure invocations:

```
add-belief create-belief "agent" 3
add-belief create-belief "location" (list 3 7)
add-belief create-belief "agent" 5
```

will result to:

```
[ ["agent" 5] ["location" [3 7]] ["agent" 3] ]
```

remove-belief [bel]
(procedure)

Removes a belief from the list of beliefs.

In the previous case if `bel = ["agent" 5]`

```
remove-belief bel
```

will result to a change in the belief structure, as shown below:

```
[ ["location" [3 7]] ["agent" 3] ]
```

exists-belief [bel]
(reporter)

Returns true if a specific the belief *bel* belongs to the set of beliefs, otherwise returns false.

For example if the beliefs are as above

```
exists-belief ["agent" 3] = true
```

exist-beliefs-of-type [b-type]
(reporter)

Reports true if a belief in the form of ["b-type" <any-content>] exist in beliefs list, otherwise returns false.

For example if the beliefs are as above

```
exist-beliefs-of-type "agent-location" = false  
exist-beliefs-of-type "agent" = true
```

beliefs-of-type [b-type]
(reporter)

Returns all beliefs of *b-type* in a list.

For example in the case above where the beliefs list is

```
[ ["agent" 5] ["location" [3 7]] ["agent" 3] ]
```

The following command will have the result indicated:

```
beliefs-of-type "agent" = [ ["agent" 5] ["agent" 3] ]
```

get-belief [b-type]
(reporter)

Returns the **first belief** of a certain type **and removes it from the beliefs list**.

read-first-belief-of-type [b-type]
(reporter)

Reports the first belief in the beliefs list (structure) that is of type *b-type* **without** removing it.

update-belief [bel]
(procedure)

Removes the first belief that is of the same type with belief *bel* and replaces it with *bel*.

Note

The code provided does not claim to be complete. However, it offers a good simulation of belief management that will help the user to experiment with various problem parameters in a multi-agent problem. Various extensions might be required-please feel free to modify the library at will and let me know!

Error handling was kept to an absolute minimum in order not to "waste" system resources.

Implementing Pro-active Agents in NetLogo

Introduction

This paragraph describes an initial attempt to implement proactive agents in NetLogo. The agents follow a PRS-like model, i.e. a set of intentions (goals) are pushed into a stack from where the agent executes them. Of course the implementation is far from delivering all the features of systems like JAM, but still can be used in implementing simple BDI agents in the NetLogo simulation platform.

The main concept behind the present implementation is the intention stack (variable `intentions` in NetLogo). This stack is used to store all intentions of the agent. Intentions are pop-ed from the stack and are executed (NetLogo command `run`) until their condition (`done`) is met. If the latter evaluates to true the intention is removed and at a next cycle the next intention is pop-ed. If the intention stack is empty then the agent does nothing. The previous behaviour is encoded in the procedure `execute-intentions`.

An intention is a NetLogo list of two elements. The first is called the intention name and maps to a NetLogo procedure and the second is the intention-done part and maps to a NetLogo reporter. For example the following intention:

```
["move [23 23]" "at-gate 3"]
```

states that the agent is currently committed to moving towards the point (23, 23) and it will retain the intention until the reporter `at-gate 3` evaluates to true. Note that the user has to specify (in NetLogo) both the procedure and the reporter, that map to the two parts of the intention.

The next paragraphs provide brief descriptions of the available procedures and reporters.

`execute-intentions`

The procedure executes an intention from the intention stack of the agent as explained above.

`get-intention`

This reporter reports the current intention of the agent, i.e. returns the list with the intention name and intention done parts.

`intention-name [intention]`

Reports the intention name (the executable procedure) of the `intention`.

intention-done [intention]

Reports the done part (argument) of the `intention`.

remove-intention [intention]

Removes a specific intention from the intention stack

add-intention [name done]

The procedure adds an intention in the intentions list. The first argument is the intention name that should be some executable procedure you encode in NetLogo. The second argument should be a reporter that when evaluates to true the intention is removed (can be used for either removing an intention when is accomplished or dropped). For example:

```
add-intention "move [23 23]" "at-gate 3"
```

BOTH ARGUMENTS HAVE TO BE STRINGS (see `run/runresult` primitive procedures in NetLogo). So you might find very useful the `word` primitive of NetLogo.

REMEMBER that intentions are stored in a STACK! For example, consider the following code:

```
...
add-intention "check-cargo" "true"
add-intention "load-cargo" "true"
...
```

The intention `load-cargo` will be executed before the intention `check-cargo`.

There is one limitation that concerns strings as arguments in intentions. If you are to add an intention that maps to an executable procedure that takes strings as arguments that you must explicitly add the quotes by inserting `\` in the appropriate places. (limitation caused by the functional language of NetLogo). For example if an intention with name `move 3` with a done part `at-destination "OA"` has to be added to the list of intentions, then the following code is required.

```
add-intention (word "move 3") (word "at-destination \"OA\" ")
```

The above applies also for strings located inside lists.

Note

The code provided does not claim to be complete with respect to what might be required to build a full BDI agent. However, it offers a good simulation platform for proactive agents. Various extensions might be required-please feel free to modify the library at will and let me know!

Error handling was kept to an absolute minimum in order not to "waste" system resources.

Have fun with NetLogo.