An Attempt to Simulate FIPA ACL Message Passing in NetLogo

Ilias Sakellariou
March 2004, updated March 2008; June 2010

Introduction

This document describes an initial version of message passing facilities for the NetLogo platform. This primitive library contains a set of necessary procedures and reporters for message creation and exchange in the platform.
In order to closely follow the FIPA ACL message format, messages are lists of the form

```
[<performative> sender:<sender> receiver:<receiver> content: <content>..]
```

For example, the following message was send by agent (turtle) 5 to agent (turtle) 3, its content is "where is the plane" and the message performative is "inform".

```
["inform" "sender:5" "receiver:3" "content:" "where is the plane?"]
```

As it can be seen from the above, agents are uniquely characterized by an ID (number) that is in fact the value of their "`who`" variable automatically assigned at their creation by NetLogo. This naming was adopted since it greatly facilitates the development of the message passing facilities.
Incoming messages for each agent are stored in a variable named `incoming-queue`. This is a "user-defined" variable that each agent must have in order to be able to communicate, i.e. defined by a `<breed>-own` declaration. Message passing is asynchronous: sending a message to agent A simply means adding (append) the message to the `incoming-queue` list; it does not require an explicit receive message command to be invoked on the receivers side. At any time the agent has the ability to obtain the messages from its queue using the reporters described below.

Since NetLogo version 4.0 supports include file facilities the code described in this document is located in a file named *communication.nls* that should be included in your model through an appropriate "`__includes`" NetLogo command (see NetLogo Manual). Apart from the above the requirements are

- All agents that are able to communicate MUST have a declared -own variable **`incoming-queue`**. This is the variable to which all messages are recorded. So, in your model if there is a breed of turtles which you desire to communicate, then you should have a **`BREED-own [incoming-queue]`** declaration (along with any other variables that you wish to include in your model. Make sure that when you create the variables you set its values to empty list (**`[]`**).

- Your model should include a "switch" named "**`show-messages`**". This is necessary since the code of the receive message checks in each cycle whether to output the messages or not.

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

Creating Messages

`create-message [performative]`

This reporter creates a new message and adds automatically the `sender:` field in the new message. The sender field contains the ID (number) of the turtle (agent) that calls the reporter (recall that every turtle has a unique ID). The argument `<performative>` contains the performative of the message and should be a string. For example, the following command:

```
set out_msg create-message "inform"
```

sets the value of the variable `out_msg` to:

```
["inform" "sender:5"]
```

`create-reply [performative msg]`

The reporter `create-reply` constructs a new message as a reply to the previous message `msg` received by the agent that calls the reporter. The `receiver:` field in the newly created message is automatically copied by the received message, so there is no need to add it explicitly. The `sender:` field is also added automatically as in the case of the `create-message` reporter. For example if the `out_msg` is

```
["query-if" "sender:5" "receiver:8" "content:" "free?"]
```

the following call

```
set repl_msg create-reply "inform" out_msg
```

will assign to variable `repl_msg` the value

```
["inform" "sender:8" "receiver:5"]
```

Adding fields to messages

After a message is created with the reporters mentioned above, appropriate fields can (have) to be added before it is send. This is done by the use of reporters as indicated below.

`add-sender [sender msg]`

Returns a message that is the original msg that appears in its second argument with the addition of a "`sender:<sender>`" field in it. This is rarely used since when a message is created this filed is automatically added. It is described here for completeness.

add-receiver [receiver msg]

Returns a message that is the original `msg` that appears in its first argument with the addition of a "`receiver:< receiver >`" field in it. For example, suppose that agent 8 executes the following code:

```
...
      let somemsg create-message "inform"
      set somemsg add-receiver 5 somemsg
```

The code above will assign to the local variable `somemsg` the message

```
["inform" "sender:8" "receiver:5"]
```

The same effect can be achieved by a call to

```
set somemsg add-receiver 5 create-message "inform"
```

(Note by author: Functional programming can be so nice!)

The `add-receiver` reporter can be called multiple times on a message, in which case it adds multiple "`receiver:`" fields. Alternatively the user can use the `add-multiple-receivers` reporter described below.

add-multiple-receivers [receivers msg]

Same as add-receiver but adds multiple receivers to a message. Note that the `receivers` argument should be a valid NetLogo list.

add-content [content msg]

Returns a message that is the original `msg` that appears in its second argument with the addition of a "`content:< content >`" field in it. The content can be anything (string, list integer). For example, the code (assumed to be executed by agent 8)

```
      let somemsg create-message "inform"
      set somemsg add-receiver 5 somemsg
      set somemsg add-content "pl ok" somemsg
```

will assign to the local variable `somemsg` the message

```
["inform" "sender:8" "receiver:5" "content:" "pl ok"]
```

Obviously the above message construction can be also achieved by the code:

```
set somemsg add-content "pl ok" add-receiver 5 create-message "inform"
```

to-report add [msg field value]

This "primitive" reporter adds a field to a message. It is used by all the reporters mentioned above. Its can also be used to add any new fields that might be considered necessary.

Accessing Information from Messages

Once a message is received the user can apply the reporters described below to access the various fields of the message. A point to note is that all reporters except the `get-content` below return **strings**. The `get-content` reporter returns exactly what it was send (i.e. string, list, integer, etc).

`get-performative [msg]`

The `get-performative` reporter returns the performative of the message `msg`. The latter is always the first item in the message (list). For example if `somemsg` is a message of the form

```
["inform" "sender:8" "receiver:5" "content:" "plane ok"]
```

then the call to

```
get-performative somemsg
```

will return `"inform"`.

`get-sender [msg]`

The `get-sender` reporter returns the sender of the message `msg`. There is always one sender for each message. For example if `somemsg` is a message of the form

```
["inform" "sender:8" "receiver:5" "content:" "plane ok"]
```

then the call to

```
get-sender somemsg
```

will return "8".

`get-content [msg]`

The `get-content` reporter returns the content of a message `msg`. The content returned is a string. For example if `somemsg` is a message of the form

```
["inform" "sender:8" "receiver:5" "content:" "plane ok"]
```

then the call to

```
get-content somemsg
```

will return "`plane ok`". If the original message was

```
["inform" "sender:8" "receiver:5" "content:" [plane 32 ok]]
```

the same call would return `[plane 32 ok]` (a NetLogo List).

get-receivers [msg]

The get-receivers reporter returns the ***list*** of receivers of a message `msg`. Note that there might be multiple receivers in a message, in which case there are multiple "`receiver:<receiver>`" entries in the message (for convenience). For example if `somemsg` is

```
["inform" "sender:5" "receiver:1" "receiver:2" "receiver:3" "content:"
"ok"]
```

then the call to

```
get-receivers somemsg
```

will return `["1" "2" "3"]`. If the message was

```
["inform" "sender:5" "receiver:1" "content:" "ok"]
```

then the same call would return `["1"]` (always a list).

Sending and Receiving Messages

This section describes how messages can be send and received using the available procedures in the library.

send [msg]

This procedure is used to send a message to other agents. There is no need to specify a receiver since the latter it is included in the message. The rule that applies is "one man's send is another man's receive - check the code for this). For example the code below (executed by agent 8-as usual):

```
let somemsg create-message "inform"
set somemsg add-receiver 5 somemsg
set somemsg add-content "pl ok" somemsg
send somemsg
```

will send to agent 5 the message that follows

```
["inform" "sender:8" "receiver:5" "content:" "pl ok"]
```

receive [msg]

The procedure is invoked by the send procedure given above. Message reception deals with updating incoming-queue, i.e. appending the msg to this list. This is **not to be used by the user**; for message reception see `get-next-message` reporter described below.

get-message

This reporter is used for obtaining a message from the message list. Its returns the first message in this list and **removes** it from the incoming-queue of the agent. If the incoming-queue is empty

then the reporter returns "no_message" For example, suppose that the incoming-queue variable of agent 5 contains the following messages

```
[     ["query-if" "sender:4" "receiver:5" "content:" "free?"]
      ["inform" "sender:10" "receiver:5" "content:" "pl bad"]
      ["request" "sender:3" "receiver:5" "content:" "unload-aircraft"]
      ["query-if" "sender:14" "receiver:5" "content:" "down?"]
]
```

the code below assigns to variable message of the agent the first message in the queue and removes it.

```
      let message get-message
```

i.e message will be:

```
["query-if" "sender:4" "receiver:5" "content:" "free?"]
```

and the incoming-queue

```
[     ["inform" "sender:10" "receiver:5" "content:" "pl bad"]
      ["request" "sender:3" "receiver:5" "content:" "unload-aircraft"]
      ["query-if" "sender:14" "receiver:5" "content:" "down?"]
]
```

Subsequently the agent can process the message with the reporters mentioned above. For example the following code, obtains a message from the queue and creates an appropriate reply:

```
      let message get-message
      let content get-content message
      let performative get-performative message
      if content="pl ok" and performative = "query-if" [
            set out create-reply "inform" message
            set out set-content "yes" out
            send out
            ]
...
```

get-message-no-remove

Same as above but does not removes the returned message for the incoming-queue.

remove-msg

This procedure removes the first message from the message list of the agent (incoming-queue), i.e. it is an explicit remove.

Broadcasting messages

`broadcast-to [breed msg]`

The procedure broadcasts to all agents of the breed `breed` the message `msg`. It is advisable that the message does not contain any other receivers added by an explicit `add-receiver` procedure call, since in this case the explicitly added receiver will get N identical messages (as many as the number of buses in the model). For example the following code with send the message "`pl ok`" to all buses in the model.

```
let somemsg create-message "inform"
let somemsg add-content "pl ok" somemsg
broadcast-to buses somemsg
```

Keep in mind that the incoming-queue is a NetLogo list of lists, and thus apart from what is given above, you can also access messages by NetLogo list primitives.

Final Note

The code provided does not claim to be complete with respect to the FIPA ACL specification. However, it offers a good simulation of message exchange 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!

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

Please report bugs to iliass@uom.gr

Have fun with NetLogo.