Adding Distribution and Fault Tolerance to Jason

Álvaro Fernández Díaz Clara Benac Earle Lars-Åke Fredlund
Babel group, DLSIIS, Facultad de Informática, Universidad Politécnica de Madrid
{avalor,cbenac,fred}@babel.ls.fi.upm.es

Abstract

In this paper we describe an extension of the multiagent system programming language Jason with constructs for distribution and fault tolerance. The standard Java-based Jason implementation already does provide a distribution mechanism, which is implemented using the JADE library, but to use it effectively some Java programming is often required. Moreover, there is no support for fault tolerance. In contrast this paper develops constructs for distribution and fault tolerance wholly integrated in Jason, permitting the Jason programmer to implement complex distributed systems entirely in Jason itself. The fault tolerance techniques implemented allow the agents to detect, and hence react accordingly, when other agents have stopped working for some reason (e.g., due to a software or a hardware failure) or cannot anymore by reached due to a communication link failure. The introduction of distribution and fault tolerance in Jason represent a step forward towards the coherent integration of successful distributed software techniques, into the agent based software paradigm. The proposed extension to Jason has been implemented in eJason, an Erlang-based implementation of Jason. In fact, in this work we essentially import the distribution and fault tolerance mechanisms from the Erlang programming language into Jason, a task which requires adaptation of the basic primitives due to the difference between a process based functional programming language (Erlang) and a language for programming BDI (Belief-Desire-Intention) agent based systems (Jason).

1. Introduction

The increasing interest in multiagent systems (MAS) is resulting in the development of new programming languages and tools capable of supporting complex MAS development. One such languages is Jason [6]. Some of the more difficult challenges faced by the multiagent systems community, i.e., how to develop scalable and fault tolerant systems, are the same fundamental challenges that any concurrent and distributed system face. Consequently, the agent-oriented programming languages provide mechanisms to address these issues, typically borrowing from more mainstream frameworks for developing distributed systems. For instance, Jason allows the development of distributed multiagent systems by interfacing with JADE [4, 5]. However, Jason does not provide specific mechanisms to implement fault-tolerant systems.

MAS and the actor model [2] have many characteristics in common. The key difference is that agents normally impose extra requirements upon the actors, typically a rational and motivational component such as the Belief-Desire-Intention architecture[10, 11].

Some programming languages based on the actor model are very effective in addressing the aforesaid challenges of distributed system. Erlang [3, 7], in particular, provides excellent support for concurrency, distribution and fault tolerance. However, Erlang lacks some of the concepts, like the Belief-Desire-Intention architecture, which are central to the development of MAS.

This article forms part of a research programme to evaluate whether the BDI architecture provides useful programming paradigms which can improve the design of (non AI-based) complex distributed systems too. However, to be able to do such an evaluation we found that it was first necessary to improve the support for programming distributed systems available in Jason implementations (which is the topic of this article).

In recent work [8], we presented eJason, an open source implementation of a significant subset of Jason in Erlang, with very encouraging results in terms of efficiency and scalability. Moreover, some characteristics common to Jason and Erlang (e.g. both having their syntactical roots in Prolog) made the implementation quite straightforward. However, the first eJason prototype did not permit the programming of distributed or fault-tolerant multiagent systems.

In this paper, we propose a distribution model and a fault tolerance mechanism for Jason closely inspired by Erlang. This extension of Jason has been implemented in eJason, thus making it possible to develop complex distributed systems fully in Jason itself. Our implementation of eJason and the sample multiagent systems described in this and previous documents can be downloaded at:

$$git://github.com/avalor/eJason.git$$

The rest of the paper is organized as follows: Section 2 provides background material introducing Jason, Erlang and eJason. Sections 3 and 4 describe the proposed distribution model and fault tolerance mechanisms for Jason programs, respectively. Some details on the implementation in eJason of these extensions can be found in Section 5. An example that illustrates in detail the use of the proposed extension is included in Section 6. Finally, Section 7 presents the conclusions and future lines of work.

2. Background

In this section we briefly introduce Jason, Erlang and eJason. Some previous knowledge of both Jason and Erlang is assumed.

2.1 Jason

Jason is an agent-oriented programming language which is an extension of AgentSpeak [9]. The standard implementation of Jason is an interpreter written in Java.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Programming based on Actors, Agents, and Decentralized Control 21-22 October, Tucson, Arizona
Copyright © 2012 ACM [to be supplied]. . . $10.00

2.1.1 The Jason Programming Language

The Jason programming language is based on the Belief-Desire-Intention (BDI) architecture[10, 11] which is highly influential on the development of multiagent systems. The first-class constructs of the language are: beliefs, goals (desires) and plans (intentions). This approach allows the implementation of the rational part of agents by the definition of their "know-how", i.e., *how* each agent should act in order to achieve its goals, based on its subjective *know*ledge.

The Jason language also follows an environment-oriented philosophy, i.e., an agent exists in an environment which it can perceive and with which it can interact using so called external actions. In addition, Jason allows the execution of internal actions. These actions allow the interaction with other agents (communication) or to carry out some useful tasks such as e.g. string concatenation and printing on the standard output, among others.

2.1.2 The Java Implementation of Jason

A complete description of the Java implementation of Jason can be found in [6]. This implementation of Jason allows the programming of distributed multiagent systems by interfacing with the well-known third-party software JADE [4, 5], which is compliant to FIPA recommendations [1]. JADE implements a distribution model where the agents are grouped in agent containers, which are in turn grouped again to compose agent platforms.

The distribution of a Jason system using JADE is not transparent from the programmer's perspective as he/she must declare the architecture of the system (centralised and JADE being the ones provided by default), and, most likely, execute some actions that rely on the third-party software used to distribute the system.

The Java interpreter of Jason provides mechanisms to detect and react to the failure of a plan of an agent. However, there are no mechanisms to detect and react to the failure of an entire agent. That is, there are no constructs which permit to detect if some agent has stopped working (has died) or has become isolated. One can of course *program* a "monitor agent" to continuously interact with the agent that should stay alive according to some pre-established communication protocol, in order to detect if the monitored agent fails. However, having to program such monitors by hand is an error prone and a tedious task.

2.2 Erlang

Erlang [3, 7] is a functional concurrent programming language created by Ericsson in the 1980s which follows the actor model. The chief strength of the language is that it provides excellent support for concurrency, distribution and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. It enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

An Erlang system (see Fig. 1) is a collection of Erlang nodes. An Erlang node (or Erlang Run-time System) is a collection of processes (actors), with a unique node name. These processes run independently from each other and do not share memory. They interact via communication. Communication is asynchronous and point-to-point, with one process sending a message to a second process identified by its process identifier (pid). Messages sent to a process are put in its message queue, also referred to as a mailbox.

As an alternative to addressing a process using its pid, there is a facility for associating a symbolic name with a pid. The name, which must be an atom, is automatically unregistered when the associated process terminates. Message passing between processes in different nodes is transparent when pids are used, i.e., there is no syntactical difference between sending a message to a process in the same node, or to a remote node. However, the node must be specified when sending messages using registered names, as the

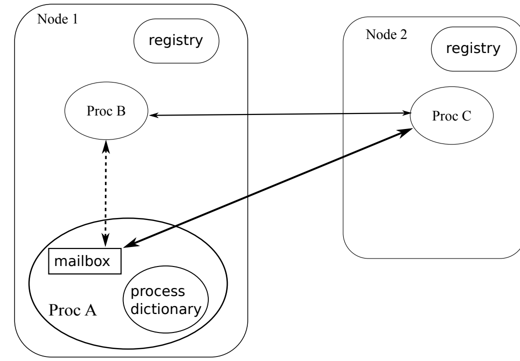

Figure 1. An Erlang multi-node system

pid registry is local to a node. For instance, in Fig. 1 let the process Proc C be registered with the symbolic name `procC`. Then, if the process Proc B wants to send a message to the process `procC`, the message will be addressed to `procC@Node2`.

A unique feature of Erlang that greatly facilitates building fault-tolerant systems is that one process can monitor another process in order to detect and recover from abnormal process termination. If a process P_1 monitors another process P_2, and P_2 terminates with a fault, process P_1 is automatically informed by the Erlang runtime of the failure of P_2. It is possible to monitor processes at remote nodes. This functionality is provided by an Erlang function named `erlang:monitor`.

2.3 eJason

eJason is our Erlang implementation of the Jason programming language which exploits the support for efficient distribution and concurrency provided by the Erlang runtime system to make a MAS more robust and performant. It is interesting to note the similarities between Jason and Erlang; both are inspired by Prolog, and both support asynchronous communication among computational independent entities (agents/processes), which makes the implementation of Jason in Erlang rather straightforward.

The first prototype of eJason was described in [8]. This prototype supported a significant subset of Jason. This subset included the main functionality: the reasoning cycle, the inference engine used by test goals and plan contexts, and the knowledge base. We continue developing eJason by increasing the Jason subset supported (which now includes plan annotation and an improved inference engine that generates matching values for the variables in the queries upon request, instead of unnecessarily computing all matching values), and by improving the design and implementation of eJason.

Probably the most relevant fact of the implementation of eJason is the one-to-one correspondence between an agent and an Erlang process (a lightweight entity), enabling the eJason implementation to execute multiagent systems composed of up to hundreds of thousands of concurrently executing agents with few performance problems. This compares very favorably with the Java based standard Jason implementation which has problems in executing systems with no more than thousands of concurrent agents (even if executed using a pool of threads) on comparable hardware (see [8] for benchmarks).

3. Distribution

In this section we describe the proposed agent distribution model extension to Jason, which has been implemented in eJason. It is inspired by the distribution model of Erlang, as, in our opinion, it is a

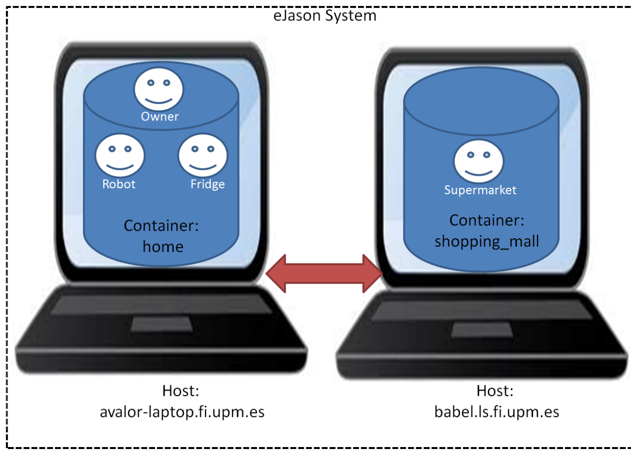

Figure 2. Sample eJason Distributed System

sound and efficient one. The distribution model has been designed with three goals in mind: distribution transparency (i.e., ensuring that distributed agents can communicate), efficiency, and minimizing the impact on the syntax of Jason programming language.

3.1 Distribution Schema

Below we introduce the terminology used to describe the distribution model.

- A **multiagent system** is composed by one or more agent containers.
- Each **(agent) container**[1] is comprised of a set of agents, and is located on a computing host. It is given a *name* that is unique in the whole system. Concretely the name has the shape `NickName@HostName`, where `NickName` is an arbitrary short name given to the container when it is started and `HostName` is the full name of the host in which the container runs. For instance, `home@avalor-laptop.fi.upm.es` corresponds to a container that runs in a host whose name is `avalor-laptop.fi.upm.es`.
- Each **agent** is present in a single container. At the moment of agent creation, it is given a symbolic name that is unique within the container. A single agent is uniquely identified in the system by the combination of its own symbolic name and the name of its container. Using the unique name of an agent, agents can communicate with each other irrespectively of the containers they reside in.

As an example, consider the distributed Jason multiagent system in Figure 2. This system is composed by four agents distributed over two different hosts. The agents with the names `owner`, `robot` and `fridge` reside in a container named `home@avalor-laptop.fi.upm.es`. The agent `supermarket` runs in the container `shopping_mall@babel.ls.fi.upm.es`, located on a different host.

3.2 Distribution API

Recall that Jason agents communicate with each other using internal actions. Thus, to support distribution, we add support for communicating in existing internal actions such as, e.g., `.send`, as well as adding a few new internal actions. In practice, most of the new internal actions are just an extension of existing ones to allow the inclusion of the name of the container where the agent receiving the effect of the internal action (e.g. receiving a message) runs. This name is added as an annotation. A complete example showing how the new API is used in practice is included in Section 6.

3.2.1 Agent Communication

The communication between agents in Jason requires the use of the internal action

a) `.send(AgentName,Performative,Message [Reply,Timeout])`,

where the parameters in brackets are optional and `AgentName` is the symbolic name of the agent receiving the message. The parameters `Reply` and `Timeout` can only be used when the performative is of type `ask`. We omit the exact description of their meaning from this article as they are present with the same semantics in "non-distributed Jason"; see [6] for details.

In the Java implementation of Jason, the symbolic name must uniquely identify one agent within the system, while, as mentioned above, the distributed extension guarantees only that agent names are unique in a container. Thus the above internal action can be used only to communicate between agents located in the same container (intra-container communications).

Therefore, to permit communication between agents located in different containers (inter-container communication) the distributed extension of Jason provides a new internal action

b) `.send(Address,Performative,Message, [Reply,Timeout])`

where the parameter `Address` is an annotated atom with the structure `AgentName[container(Container)]` and `Container` is the name of the container in which the agent with symbolic name `AgentName` runs. Most internal actions similarly accept such an address structure to specify an agent; in the following we will not list these variant internal actions.

The guarantees[2] provided by the aforementioned internal actions differ:

- If the execution of **a)** succeeds, the reception of the message by the receiver is guaranteed. However, it does not ensure that the receiving agent considers the message socially acceptable (i.e. considered suitable for processing, cf. not socially acceptable messages are automatically discarded and do not generate any event, see [6]) nor that it has a plan which can be triggered by the reception of the message. The execution will fail if there is no agent in the same container whose symbolic name is `AgentName`.
- The execution of **b)** always succeeds. This internal action, thus, provides no guarantees regarding the successful delivery of the message.

3.2.2 Agent Creation and Destruction

Agents can *create* agents in a named container using the internal action

- `.create_agent(AgentName,Source,[InitialBeliefs])`

where the parameter in brackets is optional. If `Container` is not provided, using an `Address` structure, the new agent is created in the same container as the agent executing the internal action. As in [6], the parameter `Source` indicates the implementation

[1] we use the name container to emphasize the similarities with a JADE container

[2] We encourage the reader to read the content at http://www.erlang.org/faq/academic.html#id54296 to get a feel of how useless and inefficient the imposition of strong guarantees on message delivery can be for distributed systems.

of the agent (its plans, initial goals and initial beliefs). Finally, InitialBeliefs is a list of beliefs that should be added to the set of initial beliefs of the new agent upon creation. This internal action will succeed if (1) the named container exists in the system, and (2) there is no other agent with symbolic name AgentName already in Container, and (3) Source correctly identifies an agent implementation.

An agent can also explicitly *kill* other agents. The following internal action allows this:

- .kill_agent(AgentName).

The parameters and the meaning of their absence are analogous to those above. The execution of this internal action does not fail, even if the agent to terminate does not exist.

3.2.3 Container Name Discovery

An agent can *discover* the name of its own container by executing the internal action

- .my_container(Var)

If the variable Var is unbound, the execution of the internal action does not fail and, as a result, Var will be bound to the name of the container of the agent. If Var is bound to any value different from the name of the container of the agent executing it, the internal action will fail, otherwise it succeeds.

An agent can *discover* the name of the agent and container (the complete address) from which it has received a message. Every belief and goal generated from a communication is labeled with the annotation

- source(AgentName[container(Container)])

Notice that this extension guarantees backwards compatibility with Jason legacy code. The arity of the annotation source, see [6], is maintained and the Address structure can be used to identify the sender agent, e.g. in a .send internal action that represents the reply to the message received.

4. Fault Tolerance

The fault detection extension that we describe in this section enables an agent to express its desire to be notified when another agent terminates or becomes unreachable. As we describe later, this notification is carried out by the runtime system by adding a new belief to the belief base of the agent being notified.

4.1 Agent failures

The reasons why an agent may stop working are numerous. For instance, the host in which the agent runs has been shut down, the agent itself has been stopped or has crashed due to some error in the source code, or the host in which the agent runs may have become isolated from the rest of agents in the system.

4.2 Monitoring agents

Agents are not informed about failures in each other by default. Instead, a (monitoring) agent interested in the state of a (monitored) agent must explicitly request to be notified when the status of the monitored agent worsens. This request is implemented as the internal action:

- .monitor(AgentName)

where the parameter AgentName is the symbolic name of the monitored agent in the desired container. This action is executed by the monitoring agent. The execution of this internal action never fails.

After the execution of the .monitor internal action, the monitoring agent will be informed about failures in the monitored agent at most once, i.e. after one error in the monitored agent is detected. Nevertheless, the .monitor internal action can be called again after the notification. If the monitored agent fails, the monitoring agent will receive the following new belief:

- +agent_down(AgentName[container(Container)])
[reason(RType)]

along with the corresponding belief addition event. The possible values of RType are unknown_agent, dead_agent and unreachable_agent. Their meaning is as follows:

- **unknown_agent**. There is no agent whose symbolic name is AgentName in the container Container as specified in the invocation of the internal action .monitor.
- **dead_agent**. The monitored agent with the symbolic name AgentName has stopped working.
- **unreachable_agent**. The containers of the monitored and monitoring agent are not connected, e.g. caused by a network problem, or by a problem with the container host, or by a failure in the container itself. The reconnection of the containers, which renders the monitored agent reachable again if it is still alive, may happen but it is not notified to the monitoring agent.

Clearly these errors are not mutually exclusive. For instance, an unreachable agent (**unreachable_agent**) may be dead as well (**dead_agent**).

In Section 6 we illustrate how this fault detection mechanism is used to detect and help recover from different agent failures.

5. Implementation

In this section, we provide some details about the implementation in eJason of the proposed extensions to Jason. We do not intend to give a low-level description of how every single element has been implemented. Instead, we describe the correspondence between the elements introduced and their Erlang counterparts (recall that the extensions are inspired by the distribution and fault tolerance mechanism of Erlang). A very basic knowledge about Erlang constructs and their semantics suffices to understand the contents of this section.

5.1 Distribution

The new concept introduced by our distribution model is the agent container. It is implemented using an Erlang node. Therefore, each container must be given a symbolic name unique in its host, as it is not possible to have two Erlang nodes with the same name running in the same host (Erlang nodes can be given short- or full-names. For convenience, eJason only considers the latter possibility). For each agent container in a system, a new Erlang node is started. Besides, in eJason each agent is represented by a different Erlang process. Therefore, given an agent with symbolic name AgentName running in a container with symbolic name NickName in host Host, there exists an Erlang process running in the Erlang node NickName@Host. This process is locally registered as AgentName.

The guarantees for the message exchange achieved by executing the internal action .send, described in Section 3, derive from the Erlang semantics of their implementation:

- When internal action .send(AgentName,Performative, Message,[Reply,Timeout]) is executed, a message is sent to the Erlang process locally registered as AgentName. This operation fails if there is no process registered as AgentName.
- When the internal action executed is .send(Address, Performative,Message,[Reply,Timeout]), where the parameter Address is the annotated atom described in Section 3.2.1, a message is sent to the Erlang process that runs

in the Erlang node with name `Container` and that is registered as `AgentName` in that same node. This operation cannot fail even if the Erlang node does not exist (which does not ensure the correct delivery of the message).

5.2 Faul Tolerance

The internal action `.monitor` is implemented by the Erlang function: `erlang:monitor`. Consider the case where an agent `A` monitors another agent `B`. When `A` executes the `.monitor` internal action, the Erlang process corresponding to `A` invokes the function `erlang:monitor` giving among its parameters the identifier of the Erlang process corresponding to `B` (either its registered name alone or along with the name of its Erlang node).

If there is a failure on the agent `B`, the Erlang process of agent `A` receives a so-called message 'DOWN' message, which provides, among others, information about the failure. Depending on that information, which is represented by an Erlang construct (atom or tuple), one of the different failures considered is generated:

- If the information received is the atom `noproc`, the Erlang process corresponding to the agent `B` cannot be found while the Erlang node it should be running in can. Therefore, a failure of type `unknown_agent` is detected.
- If the information received is the atom `noconnection`, the Erlang node corresponding to the container of agent `B` cannot be found. A failure of type `unreachable_agent` is detected.
- The reception of any other kind of information (e.g. the atom `killed` meaning that the Erlang process of agent `B` has been killed or a tuple providing information about the reason why that process has crashed) means that the monitored agent has died. Therefore, a failure of type `dead_agent` is detected.

6. Example

In this section we illustrate the Jason extensions using a sample multiagent system that is distributed and which makes use of the fault tolerance mechanisms described in Sections 3 and 4. The example is inspired by a similar one provided with the Jason distribution and also described in [6]. The example runs unchanged under eJason.

6.1 The system

The system is composed by the following agents:

- **Agent owner:** this agent monitors a robot (another agent) and is continuously avid for beer. It asks the robot for cans of beer. If it gets a beer, it drinks it whole sip by sip, then asks for another beer. If the robot agent reports that there are no more beers in the fridge, the owner takes a short nap and, immediately after waking up, starts asking for a beer again. Upon reception of a message from the robot informing that it is not allowed to give any more beer to the owner, it disconnects the robot. The owner monitors the robot, so if it dies (because the owner disconnects it or otherwise), the former is immediately aware of it and starts the robot again. Finally, if this agent drinks more beer than its physical limit can bear, it collapses (the agent owner dies).
- **Agent robot:** this agent fulfills the owner's requests for beer. Upon reception of a request from the owner, it goes to the fridge and checks whether it contains some beer or is empty. In the first case, it grabs one beer, goes to the location of the owner and gives him the beer. The second case is again split in two, depending on whether the supermarket (represented by another agent) is open (the supermarket agent is not dead and is reachable) or closed (the supermarket agent is dead or is unreachable). If the fridge is empty and the supermarket is open, the robot orders some beer, which will be delivered directly to the fridge, and tells the owner that the fridge is empty. Otherwise, if the fridge is empty and the supermarket is closed, the robot just tells the owner that the fridge is empty, without trying to make any order. In order to know whether the supermarket is open or not, the robot monitors the supermarket agent. Depending on the type of failure detected in the supermarket agent, the robot emits a different speech (prints a different message on the standard output). Finally, the robot also monitors the owner and, if it dies, emits a short speech and waits, idly, for future requests.
- **Agent fridge:** this agent receives requests for beer from the robot. If it is not empty, it gives a beer to the robot, hence decrementing its current stock by one unit. If it is empty, it does not hand out any beer to the robot and just reports back this fact. Finally, anytime it receives a delivery from the supermarket, it updates its contents.
- **Agent supermarket:** the behaviour of this agent is the simplest in the system. It waits for a delivery order from any agent (the robot in this case) and fulfills it.

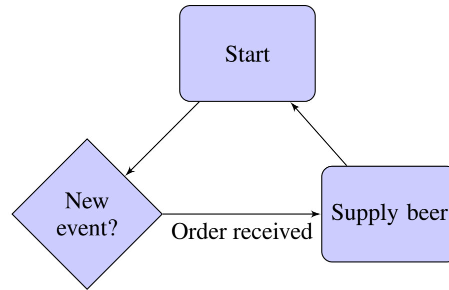

Figure 3. Flowchart for agent "supermarket"

The eJason code for each of these agents is included in Appendix A, Figures 7, 8, 9, and 10.

For the sake of clarity, we also include a series of flowcharts in Figures 3, 4, 5, and 6 that provide the diagrammatic representation of the behaviour of each of the agents, respectively.

Note that the environment entity is not implemented in eJason yet, hence not allowing the execution of external actions or the gathering of information through perception. Therefore, some actions of the system, like the displacement of the robot or checking the contents of the fridge, have been assumed to be always successful and do not require any interaction with the environment.

6.2 Code Excerpts

In this section we provide some brief code excerpts showing agent plans, to illustrate the new features introduced in Jason (and implemented in eJason).

6.2.1 Communication

Consider the following plan from the agent robot, where both intra- and inter-container communication take place:

```
+no_more(beer) :                          //Trigger
    (not  closed(supermarket)) &          //Context
    address(supermarket,SupContainer) <-  //_____
   -no_more(beer);                        //Body
   .print("Fridge is empty.");            //
   .send(supermarket                      //
        [container(SupContainer)],        //
        achieve, order(beer,5));          //
   .send(owner,tell,no_more(beer));       //
```

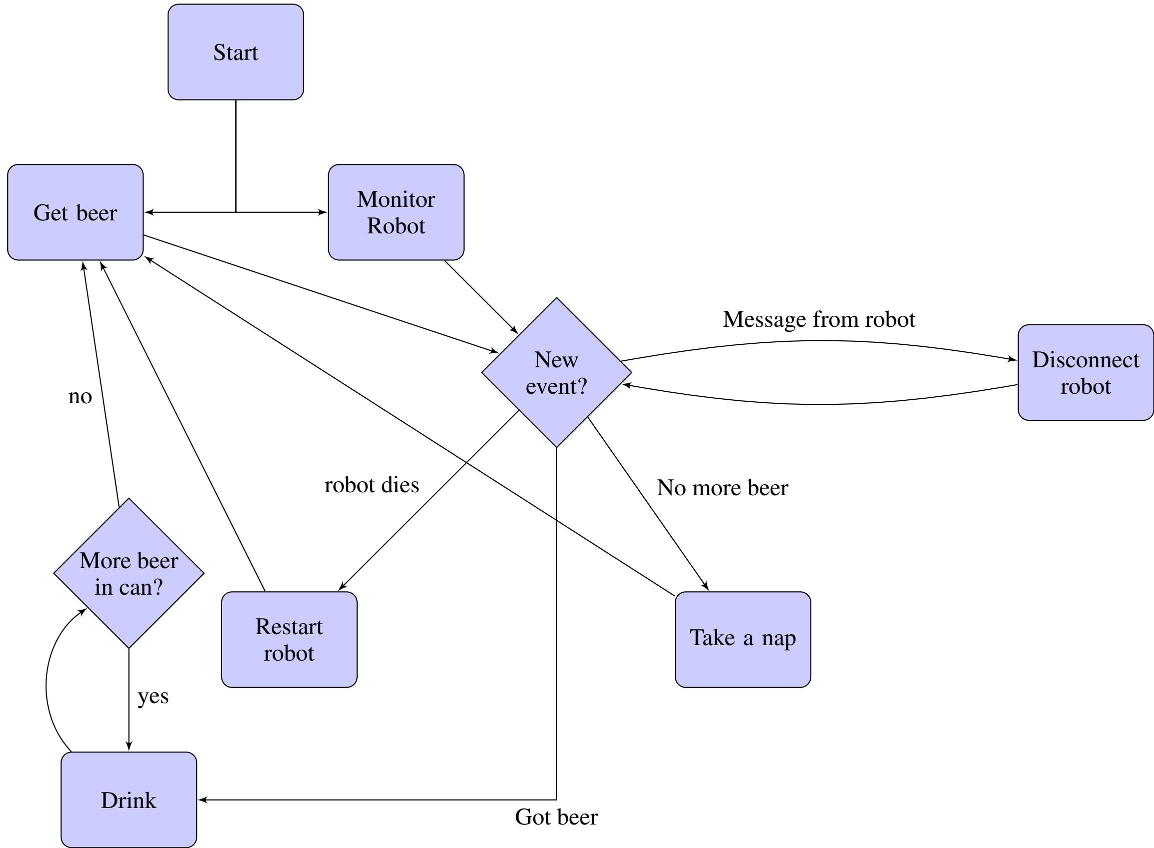

Figure 4. Flowchart for agent “owner”

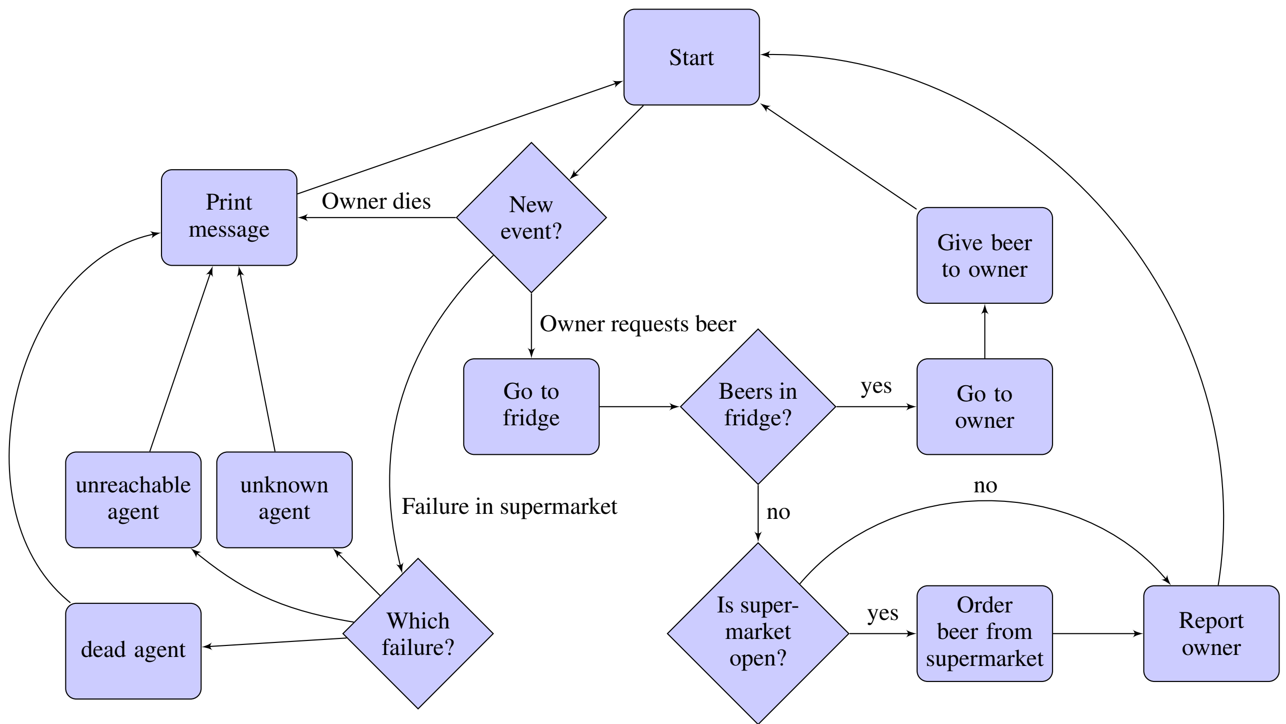

Figure 5. Flowchart for agent “robot”

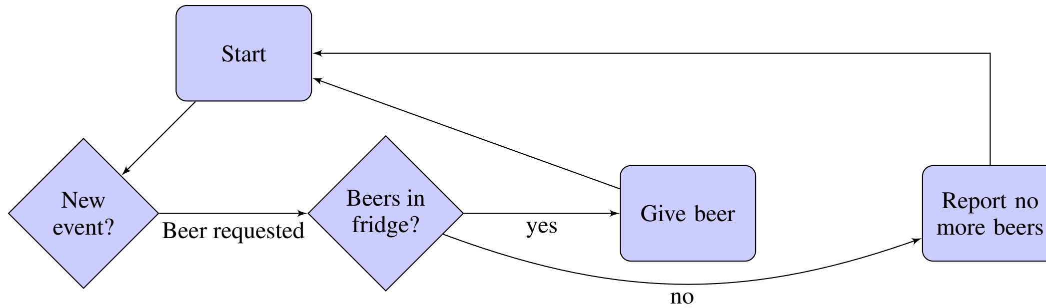

Figure 6. Flowchart for agent "fridge"

```
   !at(robot,fridge).                        //_____
```

This plan is triggered when the robot notices that the fridge is empty. The context requires the supermarket not to be closed (i.e. the agent supermarket must be alive and reachable) and the address of that agent to be known. When the plan is triggered and the context is met, the robot executes, in appearance order, the actions in the body of the plan. The first action deletes the belief that states that the fridge is empty while the second makes the robot print a message on the standard output. The third action is a `.send` action of type `b)` according to Section 3.2.1, where `AgentName` is `supermarket`, `Container` has the value of the variable `SupContainer` (which in this case is `shopping_mall@babel.ls.fi.upm.es`), `Performative` is `achieve` and `Message` is `order(beer)`. By executing this action, the robot is ordering beer from the supermarket agent, which runs in a different container. The fourth action is a `.send` action of type `a)` and does not include any reference to the container in which the agent owner runs, as it is the same one for robot agent. Finally, the last action represents an achievement goal that requires the robot to go to the location of the owner.

6.2.2 Monitoring an agent

The code for the agent robot also shows how an agent can monitor another agent. Consider the following excerpt:

```
+!monitor(Agent):
    address(Agent,Container) <-
   -closed(supermarket);
   .monitor(Agent[container(Container)]).

+!monitor(owner): true <-
   .monitor(owner).
```

The first plan can only be executed if the full address of the monitored agent is known.

The second plan will only be executed if the monitored agent is the owner agent. Notice that, this time, the container of the monitoring and monitored is the same, hence being omitted from the parameters of the `.monitor` action.

6.2.3 Detecting different failures

Finally, consider these four plans, again from the source of the robot agent:

```
+agent_down(supermarket)
     [reason(unknown_agent)]: true <-
   +closed(supermarket);
   .print("I cannot find the supermarket
           in the shopping mall").

+agent_down(supermarket)
     [reason(unreachable_agent)]: true <-
    +closed(supermarket);
    .print("I cannot find the shopping mall").


+agent_down(supermarket)
     [reason(dead_agent)]: true <-
    +closed(supermarket);
    .print("The supermarket has just closed").


+agent_down(owner): true <-
   .print("Oh, oh. My master has passed out.").
```

The first plan is triggered when a failure of type `unknown_agent` is detected on the supermarket agent. This can only happen if the container `shopping_mall@babel.ls.fi.upm.es` is reachable but does not contain any supermarket agent. A possible reaction from the robot agent could have been starting that agent, but we did not consider reasonable the option of allowing the robot to "open" the supermarket.

The second plan is only triggered when a failure of type `unreachable_agent` is detected on the supermarket agent. This failure is generated when the connection to the container named `shopping_mall@babel.ls.fi.upm.es` is lost, hence leaving the agent supermarket unreachable.

The third plan is triggered when a failure of type `dead_agent` is detected again on the supermarket agent, i.e. the Erlang process corresponding to that agent is dead. Again, a possible reaction from the robot agent could have been starting that agent.

The fourth plan is triggered by the detection of any kind of failure on the owner agent, as the annotation of the event is ignored. These four plans represent all the possible agent failure detections that are possible in the example.

Notice that, in all four plans, the `[container(Container)]` annotation corresponding to each `AgentName` has been ignored in their triggers, as this information is not used neither in their contexts nor in their bodies.

6.3 Experiments

In this section we report on some experiments that we performed on the multiagent system described above. In all of them, the distribution of the agents was organized in the following way, which is also depicted in Figure 2:

- The agents owner, robot and fridge run in the same container whose name is *home@avalor-laptop.fi.upm.es*.

- The agent supermarket runs in a different host and container. The name of this container is *shopping_mall@babel.ls.fi.upm.es*. Checking Figure 10, one may notice that this address is hard-coded as an initial belief of the robot agent. It is done this way for convenience, as no service discovery is available for the agents yet.

The experiments carried out were the following:

6.3.1 Experiment 1: distribution

The goal of this experiment is checking whether the implementation of the proposed distribution works correctly. To do it, we start all the agents of the system but do not connect the two hosts until some time after the start of the experiment. If the distribution works properly, after emptying the fridge, the owner must not get more beer until the robot can order more from the supermarket, which is not possible while the hosts remain disconnected. Next, we list the steps of the experiment (enumerated using letters) together with a description of the observable behaviour of the agents that is relevant to the experiment (presented between angle brackets).

a) Disconnect the hosts (isolating one of them suffices).

b) Start all the agents.

<The owner drinks all four beers in the fridge. The robot informs that the fridge is empty, but does not attempt to order new beers because it is aware of the fact that the supermarket is unreachable. The owner sleeps and wakes up periodically asking the robot for more beer. The system does not evolve.>

c) Connect the hosts via internet or intranet

<The robot orders more beer from the supermarket. The fridge gets refilled. The owner continues drinking.>

c) Terminate the experiment (kill all the agents).

The experiment shows that the distribution model implemented works as expected in this case. Besides, note that the two varieties of the internal action `.send`, described in Section 3.2.1, are successfully used for both intra- and inter-container communication.

6.3.2 Experiment 2: intra-container fault tolerance

This experiment seeks to test the performance of the fault tolerance mechanisms implemented when they involve agents running in the same container. Briefly, it follows an execution flow in which the agents owner and robot stop working (die) several times. Anytime the robot is not alive, the owner must notice it and start it again (even if the owner killed it). If the owner agent dies, the robot must be aware of it and print a message. The steps and output of the experiment are:

a) Connect the hosts and start all the agents but the robot.

< The owner immediately realizes that the robot agent is not alive and starts it. The owner starts drinking beer. The robot refills the fridge whenever it is empty. The sequence continues until the robot tells the owner not to drink any more beer. Upon reception of the message from the robot, the owner kills it. The owner becomes immediately aware that the robot is dead and restarts it. The owner continues drinking beer, killing and restarting the robot whenever it refuses to bring more beer, until it surpasses its physical limit and passes out (the agent dies). The robot notices the death of the owner immediately after it happens and prints a message.>

b) Terminate the experiment (kill all the agents).

Note that the failures detected in this experiment are all of type `dead_agent`. This experiment shows that the agents are immediately aware of the death of the agents they monitor, at least in the same container, and react properly (the owner restarting the robot and the robot printing a message).

6.3.3 Experiment 3: inter-container fault tolerance

This third experiment tests the performance of the fault tolerance mechanisms when the system is distributed. In its execution flow the three types of agent failures described before are generated. The correct detection of these failures can be checked through the reactions of the agent robot. If the agent supermarket is not alive but its container is, the robot must say: "I cannot find the supermarket in the shopping mall". If the agent supermarket dies, the robot says "The supermarket has just closed".Finally, if the agent supermarket becomes unreachable, the robot must say: "I cannot find the shopping mall". The steps followed and the relevant observable behaviour of the agents are:

a) Connect the hosts and start all the agents but the supermarket.

<The robot is immediately aware that there is no agent with symbolic name supermarket in the container shopping_mall and prints the message: "I cannot find the supermarket in the shopping mall". The owner drinks beer until the fridge is empty, then gets asleep and wakes up periodically.>

b) Start the agent supermarket

<The fridge gets filled again and the owner continues drinking beer.>

c) Disconnect the hosts

< The fridge becomes empty. Then, the robot tries to order beer from the supermarket for a while. After about 60 seconds after the disconnection of the hosts (due to Erlang implementation issues, the disconnection of two nodes running in different hosts is detected, by default, from 45 to 70 seconds after the network disconnection actually happened), the robot realizes that the supermarket is unreachable and prints the message: "I cannot find the shopping mall".>

d) Reconnect the hosts.

<The fridge gets filled again and the owner continues drinking beer.>

e) Kill the agent supermarket.

<The robot agent is immediately aware of the death of the agent supermarket and prints the message: "The supermarket has just closed">

f) Start the agent supermarket.

<The fridge gets filled again and the owner continues drinking beer.>

g) Destroy the container `shopping_mall`

<The robot agent is immediately aware of the death of the agent supermarket and prints the message: "The supermarket has just closed">

h) Terminate the experiment (kill all the remaining agents).

The experiment shows that all the different failures are correctly detected by the implementation of our proposed fault tolerance system.

7. Conclusion and Future Work

In this paper we have described an extension to the Jason multiagent systems programming language, which provides a new distribution model and constructs for fault detection and fault recovery. Moreover, our implementation of Jason in Erlang – eJason – contains a prototype implementation of the extension.

The addition of a proper, Jason based, distribution model to Jason systems, removes the need of interfacing to third-party software such as e.g. JADE. The addition of fault-detection and fault tolerance mechanisms to Jason addresses one of the key issues in the development of robust distributed systems. Concretely, these mechanisms allow the detection of failures in a multiagent system caused by malfunctioning hardware (computers, network links) or software (agents).

The distribution model and fault-tolerant mechanisms are inspired by Erlang, an actor based language which is increasingly used in industry to develop robust distributed systems, in part precisely because of the elegant approach to fault detection and fault tolerance. Somewhat surprising, Erlang and Jason share many common features, and this has made the design and implementation of the distribution model and the fault-tolerant mechanisms a rather straightforward task.

The lines of future work are many. First we need to provide higher-level agents (components) that ease the task of programming fault tolerant systems. In Erlang this is accomplished, for example, by providing a general component (the supervisor) to supervise and, if need be, restart failing processes. In Jason this will correspond to a monitoring agent. We expect to extend the usual notion of supervision (responding to termination of agents) with a notion of "semantic termination", i.e., detecting when an agent is still alive but no longer contributing useful results. In this same line, we plan to elaborate use cases showing how higher-level fault-tolerance properties could be implemented in eJason. Nevertheless, some issues like the preservation of the state of an agent across failures or the identification/development of useful high-level constructs must be dealt with first.

Second, we should develop at least a "semi-formal" semantics for this extension of Jason, describing exactly the behaviour of the internal actions (sending, monitoring, etc).

Third, we need to evaluate this extension on further examples, to ensure that the new internal actions have sufficient expressive power, and are integrated well enough in Jason, to permit us to design distributed multiagent systems cleanly and succinctly, and in the general spirit of BDI reasoning systems.

Considerably more speculative, we are not certain that the model of distribution offered by this extension is sufficient to model real-world multiagent system with respect to agent mobility. It might be necessary, for instance, to permit an agent to migrate between different process containers. This will complicate the implementation, but should not be impossible

With respect to eJason, there is a need to complete the implementation with regards to environment (perception of external events, etc) handling, with all its related functionality. Moreover, we plan to permit the interoperability between the agents running in eJason and agents belonging to other (e.g., JADE based) agent platforms. Therefore, we will study how to best interface eJason with a FIPA based agent platform. For instance, the concept of Directory Facilitator appears similar to the global registry service provided by Erlang.

References

[1] *Foundation for Intelligent Physical Agents, Agent Communication Language. http://www.fipa.org/specs/fipa00061/SC00061G.html.*

[2] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.

[3] J. Armstrong. Programming Erlang: Software for a concurrent world). The Pragmatic Bookshelf, 2007.

[4] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - a Java agent development framework. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, 2005. ISBN 0-387-24568-5.

[5] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Wiley, Apr. 2007. ISBN 0470057475. URL `http://www.worldcat.org/isbn/0470057475`.

[6] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007. ISBN 0470029005.

[7] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009. ISBN 978-0-596-51818-9—ISBN 10:0-596-51818-8.

[8] Á. Fernández-Díaz, C. Benac-Earle, and L.-A. Fredlund. ejason: an implementation of jason in erlang. Proceedings of the 10th International Workshop on Programming Multi-Agent Systems (ProMAS 2012), pages 7–22, 2012.

[9] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996. ISBN 978-3-540-60852-3. doi: 10.1007/BFb0031845. URL `http://www.springerlink.com/content/5x727q807435264u/`. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 Jan. 1996, Proceedings.

[10] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *In Proceedings of the first Interntional Conference on Multi-Agent Systems (ICMAS-95*, pages 312–319, 1995.

[11] M. Wooldridge. Reasoning about rational agents. MIT Press, 2000.

A. Appendix A

```
contents(beer,4).

+!give(Item)[source(Who[container(Where)])]:
    contents(Item,Stock) &
    Stock > 0 <-
   .print("Giving beer to ",Who,
          " in ", Where);
   NewStock =  Stock + -1;
   -+contents(Item,NewStock);
   .print("Beers left:  ",
          NewStock);
   .send(Who[container(Where)],tell,
         holding(Item)).

+!give(Item)[source(Who)] :
    contents(Item,Stock) &
    Stock < 1 <-
   send(Who,tell,
        no_more(Item)).

+delivered(Item,Qtd, OrderId):
    contents(Item,Stock)<-
   -delivered(Item,Qtd,OrderId);
   .print("Received ", Qtd,
          " units of ", Item);
   NewStock = Qtd + Stock;
   -+contents(Item,NewStock).
```

Figure 7. Code for agent “fridge”

```
last_order_id(1).

+!order(Product,Qtd)[source(Ag[container(Container)])]:
   last_order_id(N) &
   OrderId = N +1<-
  -+last_order_id(OrderId);
  .print("Sending ", Qtd, " units of ",
         Product, "to ", Container);
  .send(fridge[container(Container)],tell,
        delivered(Product,Qtd,OrderId)).
```

Figure 8. Code for agent “supermarket”

```
physical_limit(21).
beers_drunk(1).
inactive(robot).

!monitor(robot).

+!get(beer):
     physical_limit(Limit) &
     beers_drunk(Drunk) &
     Drunk <= Limit &
     not inactive(robot)<-
   .send(robot, achieve,
         has(owner,beer));
   .print("Getting a beer").

+!get(beer):
      physical_limit(Limit) &
      beers_drunk(Drunk) &
      Drunk >Limit &
      not inactive(robot)<-
     .print("I feel strangg..");
     .kill_agent(owner).

+has(owner,beer) : true <-
   ?beers_drunk(Beers);
   .print("I got my beer number ",
           Beers, ". Yeepe!");
   +remaining_sips(3);
   !!drink(beer).


+!drink(beer) :
     remaining_sips(Sips) &
     Sips > 0 <-
    NewSips = Sips + -1;
    .print("Sip");
    -+remaining_sips(NewSips);
    ?remaining_sips(X);
    !!drink(beer).

+!drink(beer) :
     remaining_sips(Sips) &
     Sips < 1 <-
    ?beers_drunk(Beers);
    NumBeers = Beers +1;
    -+beers_drunk(NumBeers);
    -has(owner,beer);
    .print("Finished beer!");
    !!get(beer).

+msg(M)[source(Ag)] : true <-
    .print(Ag," says: ",M);
    -msg(M);
    .print("Unacceptable!");
    .print("Let's restart my ",
           "mechanic friend.");
    .kill_agent(Ag).

+closed(supermarket): true <-
    -closed(supermarket);
    !sleep.

+no_more(beer): true <-
    -no_more(beer);
    !sleep.

+!sleep: true <-
    .print("No beer means nap ",
           "time Zzz.");
    -closed(supermarket);
    .wait(2000);
    !!get(beer).

+agent_down(robot): true <-
    +inactive(robot);
    -agent_down(robot);
    .create_agent(robot,robot);
    .print("robot has stopped ",
           "working. Start anew!");
    !monitor(robot).

+!monitor(robot):true <-
    -inactive(robot);
    .monitor(robot);
    !!get(beer).
```

Figure 9. Code for agent "owner"

```
consumed(beer,0).
at(robot,owner).
limit(beer,10).
address(supermarket,
   'shopping_mall@babel.ls.fi.upm.es').

too_much(Beverage) :-
     limit(Beverage,Limit) &
     consumed(Beverage,Consumed) &
     Consumed > Limit.

!monitor(supermarket).

+!has(owner,beer):
     not too_much(beer) <-
    !at(robot,fridge);
    .send(fridge,achieve,give(beer)).


+!has(owner,beer):
     too_much(beer)<-
    ?limit(beer,Y);
    .print("The Department of Health ",
           "does not allow me to give",
           " you more than ",Y,
           " beers a day! I am very ",
           "sorry about that!");
    ?consumed(beer,X);
    .print("Consumed ",X,
           " beers when the limit is ",
           Y, ".");
    .send(owner,tell,
          msg("I am very sorry!")).


+!has(owner, beer):
     closed(supermarket) <-
    .send(owner,tell,
          closed(supermarket));
    !monitor(supermarket).


+holding(beer) : true <-
    -holding(beer);
    !at(robot,owner);
    ?consumed(beer,X);
    Y = X +1;
    -+consumed(beer,Y);
    .send(owner, tell, has(owner,beer)).


+no_more(beer) :(not
     closed(supermarket)) &
    address(supermarket,SupContainer) <-
   -no_more(beer);
   .print("Fridge is empty.");
   .send(supermarket[container(SupContainer)],
         achieve, order(beer,5));
   .send(owner,tell,no_more(beer));
   !at(robot,fridge).

+no_more(beer):
    closed(supermarket)<-
   -no_more(beer);
   .print("Fridge is empty.
          And supermarket is closed.");
   .send(owner,tell,
          closed(supermarket));
   !monitor(supermarket);
   !at(robot,fridge).

+!at(robot,P):
    at(robot,P) <-
   true.

+!at(robot,P):
    not at(robot,P)<-
   -+at(robot,P).


+!monitor(Agent):
    address(Agent,Container) <-
   -closed(supermarket);
   .monitor(Agent[container(Container)]).

+!monitor(owner): true <-
   .monitor(owner).

+agent_down(supermarket)
     [reason(unknown_agent)]: true <-
   +closed(supermarket);
   .print("I cannot find the supermarket
           in the shopping mall").


+agent_down(supermarket)
     [reason(unreachable_agent)]: true <-
    +closed(supermarket);
    .print("I cannot find the shopping mall").


+agent_down(supermarket)
     [reason(dead_agent)]: true <-
    +closed(supermarket);
    .print("The supermarket has just closed").


+agent_down(owner): true <-
   .print("Oh, oh. My master has passed out.").
```

Figure 10. Code for agent "robot"