On Modelling Agent Systems with Erlang *

Carlos Varela and Carlos Abalde and Laura Castro and Jose Gulías
MADS Group, LFCIA Lab, Computer Science Department
University of A Coruña
Campus de Elviña s/n. 15071, A Coruña, Spain
{cvarela, carlos, laura, jgulias}@dc.fi.udc.es

Abstract

Multi-agent systems are a kind of concurrent distributed systems. In this work, some guidelines on how to create multi-agent systems using Erlang are presented. The modelled system supports cooperation among agents by plan exchange, reconfiguration and has a certain fault-tolerance. The distributed and concurrent functional programming Erlang, together with OTP platform, allows the creation of high-availability and fault-tolerant concurrent and distributed systems, and it seems to be an interesting framework for implementing multi-agent systems.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence-Multiagent systems

General Terms

Design

Keywords

Distributed Systems, Functional Programming, Multi-agent Systems

1 Introduction

Distributed systems are getting more important each day with the Internet revolution. In this environment, distribution of responsibilities is the natural way of building such systems and there is an evolution in how to program them. To deal with these systems many approaches have been proposed, for example Multiple Agent Systems. Within agents systems, BDI [1] is a very popular architecture, and we use it in our work as starting point for system modelling.

Our group has been using Erlang [2] for the last years and we think that this language can be a quite suitable implementation platform for multi-agent system programming due to its communication mechanisms, multi-thread features, fault-tolerance... In this paper, an overview of how such systems can be implemented is shown, putting emphasis on the declarative nature of the language.

The paper is structured as follows: First, a short introduction to agent systems is presented together with the BDI architecture. Then, our agent architecture is shown, gradually introducing additional features to the general model: cooperation, reconfiguration and fault-tolerance capabilities. Finally, we conclude.

2 Intelligent Agents

An agent is a computer system that is situated in some environment, and it is capable of autonomous action in the environment to meet its design objectives. In most domains, an agent does not have complete knowledge of the environment and complete control over it. The environment may evolve dynamically independently of the agent, and the actions of the agent may fail. In general, the environment is assumed to be non-deterministic. The key problem for an agent is deciding which of its actions should be performed in order to satisfy its design objectives. The main capabilities an intelligent agent should have are:

- **Reactivity:** Intelligent agents can perceive their environment, and respond in a timely fashion to changes that occur in it.
- **Proactiveness:** Intelligent agents can exhibit goal-directed behaviour by taking the initiative.
- **Social ability:** Intelligent agents are capable of interacting with other agents (and possibly humans).

Proactiveness means that the agent has a goal-directed behaviour. If the environment does not change, it is not hard to build a system which exhibits such kind of behaviour: it can be achieved with a procedure in any usual programming language. In many environments, however, the agent cannot observe it completely, there can be other agents which can change the environment, and so on. In such cases, an agent must be *reactive*, that is, it must be responsive to events that occur in the environment, since these events may alter either its goals or the assumptions which underpin the procedures in execution.

Building purely goal-directed or purely reactive systems is not too hard. What turns out to be hard is building a system that achieves

*Partially supported by MCyT TIC 2002-02859 and Xunta de Galicia PGIDT 02TIC00101T

Erlang'04, September 22, 2004, Snowbird, Utah, USA.

an effective balance between goal-directed and reactive behaviour. We do not want our agent to be reacting too often, or, on the other hand, to focus on a goal too long.

Finally, *social ability* means that an agent must be able to act in multi-agent environment by *cooperating* and *negotiating* with other agents in order to achieve its goal. This may require to reason about the goals of others, or to share the goals with other agents.

2.1 BDI model

Often agents are modelled by means of high-level cognitive specifications involving concepts such as beliefs, knowledge, desires, intentions... Here we will see the *belief-desire-intention* model of rational agency. Intuitively:

- *Beliefs* correspond to information the agent has about the world. It may be incomplete or incorrect.
- *Desires* represent state of affairs that the agent would wish to be brought about.
- *Intentions* represent desires that the agent has committed to achieve.

3 Agents' Architecture

As proposed in [3] and [4] agents are implemented as groups of communicating processes. These processes can then perform specific tasks, such as communicating with other agents or performing computations. This is the natural way to design applications in Erlang and the language influenced the basic architecture of the agents.

Architecture is based on BDI model [1]: the agent has knowledge of the world (beliefs) and a list of desires (goals) that appear on the basis of received events. Events could be internal or external events (perceptions). When an agent can achieve a goal, this becomes an intention and then it can be executed.

We choose to implement beliefs, goals and events following the same format, they are represented using Erlang tuples with the form $\{Name, Param\}$, for instance:

```
{fire,{100,10}}  % Fire on pos 100,10
{mylocation, {50,15}}  % I am on pos 50, 15
{weather, dry}
...
```

As stated at the beginning of this section, an agent is made up of processes. Initially there are four processes: an event receptor process, a state server, an executor and the main process. The agent's state is maintained on a server, which is a process that maintains the list of goals, intentions, beliefs and the agent's internal state. Together with this process, there is a process which is in charge of receiving events (messages) - here a first filtering is done and the events which the agent cannot understand (i.e, they do not have the expected form) are discarded. It may not seem necessary to use such process, but in the next sections we will see that this process will redirect the incoming messages to the appropriate process. Once processed, the remaining events which were not filtered out are finally sent to the main process. The main process performs the following tasks:

1. Receive an event
2. Update beliefs

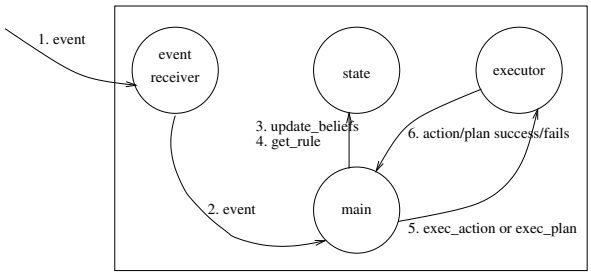

Figure 1. Agent's internal structure

3. Produce actions and/or goals
4. Execute actions.
5. A goal is selected and a plan is launched.

The Erlang code that implements the commented behaviour is (implemented using a *gen_server* behaviour):

```
handle_cast({external, {Event, Param}}, State) ->
    ok = update_beliefs(State, {Event, Param}),
    {ok, NewState} =
        case produce_actions_goals(State, {Event, Param}) of
            false ->
                {ok, State};
            true ->
                ok = execute_actions(State),
                execute_goals(State)
        end,
    {noreply, NewState}.

produce_actions_goals(State, {Event, Param}) ->
    case state:get_event_action_goal_list(
              State#state.pid_state, Event) of
        {ok, List} ->
            ok = add_actions_goals(State, Param, List),
            true;
        {error, not_found} ->
            {ok, no_answer}
    end.
```

The executor is in charge of checking the applicability of the plans and to execute them, besides checking that the invariant holds during the plan execution. It is also in charge of carrying out the actions. The internal structure and the communications among the agents can be seen in Fig. 1 Its behaviour is as follows:

1. Check the plan precondition.
2. If precondition fails then the plan is cancelled and a fail event is sent to the main process.
3. Invariant is checked.
4. If the invariant fails, plan is cancelled and a fail event is sent to the main process.
5. Execute next step of the plan.
6. If it is an action, it is executed.
7. If it is a goal, the state of the actual plan is stored and a plan to achieve the goal is found, fetched and executed.
8. If execution fails, a fail event is sent to the main process.
9. Continue executing the steps of the plan and checking the invariant until the invariant fails or the plan is finished.
10. If the plan is finished, a finished plan event is sent to the main process.

To reach a goal, an agent has to use plans. Plans specify how to

achieve goals by describing the steps to be taken. In our proposal, plans are represented with an Erlang record whose definition is:

```
-record(plan, {trigger,precondition,postcondition,invariant,body}).
```

trigger: Goal which triggers the plan execution.

precondition: Conditions that must be true in the agent's state to be able to execute the plan. It is represented as a functional value which returns a boolean value. It is evaluated before launching the plan; if the returned value is *false* the plan is not carried out.

postcondition: Conditions that must be true in the agent's state at the end of the plan. As in precondition, it is represented with a closure which returns a boolean value. If it returns *false*, the plan fails.

invariant: Condition which must be true while the plan is executing. It is a function, as in precondition and postcondition, but in this case it is executed after each step of the plan is performed. If it does not return *true*, the plan fails.

body: Sequence of actions to execute and/or goals to achieve in order to accomplish the plan. Each item in the body is an Erlang tuple which states if it is an action or a goal and, optionally, a condition to execute the plan. The fact of the condition being false does not imply that the execution of the plan fails. The atom *precondition* indicates that the precondition of the plan must be true. Condition is a function which returns *true* o *false*, or it can be a special atom like *precondition*.

As an example, consider the following: plan of a fire extinguisher robot. The plan to satisfy the goal {*put_out_fire*,{*X*,*Y*}} first plans a route to {*X*,*Y*}, next it moves to that point and launches the plan to achieve the goal {*extinguish_fire*, {*X*,*Y*}}, this new plan tries to extinguish the fire by squirting. If the fire is still there then the plan is launched again. The code that represents this behaviour is:

```
[#plan{
  trigger=put_out_fire,
  precondition=fun(State) ->
     {ok, {put_out_fire, {X,Y}}} =
         state:get_actual_goal(State#executor_state.pid_state),
     state:beliefs_contains(State#executor_state.pid_state,
                            {fire_at, {X,Y}})
  end, % Must believe that there is a fire at X,Y
  postcondition=fun(State) ->
    {ok, {put_out_fire, {X,Y}}} =
        state:get_actual_goal(State#executor_state.pid_state),
    not state:beliefs_contains(State#executor_state.pid_state,
                               {fire_at, {X,Y}})
  end, % The postcondition is the same
  invariant=fun(State) ->
     {ok, {put_out_fire, {X,Y}}} =
         state:get_actual_goal(State#executor_state.pid_state),
     state:beliefs_contains(State#executor_state.pid_state,
                            {fire_at, {X,Y}})
  end, % The invariant is the same
  body=[{action, plan_route},
        {action, move},
        {goal, {extinguish_fire,
                fun(State) ->
                  PidState = State#executor_state.pid_state,
                  {ok, {put_out_fire, {X,Y}}} =
                      state:get_actual_goal(PidState),
                  {X,Y}
                end}}]},
 #plan{
  trigger=extinguish_fire,
  precondition=fun(State) ->
    {ok, {extinguish_fire, {X,Y}}} =
        state:get_actual_goal(State#executor_state.pid_state),
    state:beliefs_contains(State#executor_state.pid_state,
                           {fire_at, {X,Y}})
    and
    state:beliefs_contains(State#executor_state.pid_state,
                           {mylocation, {X,Y}})
  end, % Must believe that there is a fire at X,Y
       % and that we are in this position
  postcondition=fun(State) -> true end, % No postcondition
  invariant=fun(State) ->
    {ok, {extinguish_fire, {X,Y}}} =
        state:get_actual_goal(State#executor_state.pid_state),
    state:beliefs_contains(State#executor_state.pid_state,
                           {mylocation, {X,Y}})
  end,
  body=[{action, squirt},
        {goal, {extinguish_fire,
                fun(State) ->
                  PidState = State#executor_state.pid_state,
                  {ok, {extinguish_fire, {X,Y}}} =
                      state:get_actual_goal(PidState),
                  {X,Y}
                end}, precondition}]}]
```

All funs on the plan (precondition, postcondition, invariant and condition of goals) receive an argument which represents the state of the *executor* and allows to obtain important values such as the Pid of the process which represents the state of the whole agent. The structure of the agent will be explained later.

As can be seen in the example, actions can return values. These values are stored on the agent's state and can be recovered to be used as parameters on other actions or to define goals.

To launch a plan the agent needs to have one or more *goals*. Goals arise from *events* and *beliefs*. Some authors do not distinguish between events and goals but, in [5] there is a clear difference between them. Events arise from the perceptions of the environment and they can imply direct actions (reflexive actions) and/or goals.

Our system uses rules to indicate what actions/goals are produced by an external event. The rules which indicate what an event produces have the following form:

```
<EventRuleList> ::= [ <EventRuleListContent> ]
<EventRuleListContent> ::= <EventRule> | <EventRuleListContent>
<EventRule> ::= { <Event> , <ActionGoalList> }
<ActionGoalList> ::= [ <ActionGoalListContent> ]
<ActionGoalListContent> ::= Goal | Action |
                            <ActionGoalListContent>
<Goal> ::= { goal , <atom> }
<Action> ::= { action, <atom> }
```

for example:

```
[{fire_at, [{goal, put_out_fire}]}]
```

An event can produce several goals and/or direct actions. The order in which actions are executed is the order in which they appear on the rule.

After presenting this core behaviour, we are going to extend it with a sort of cooperation. The kind of cooperation desired is to deal with the case when an agent receives a well-formed event but it does not know how to attend to it. This kind of cooperation is considered in the next section.

3.1 Cooperation

Sometimes, an agent can receive events which it cannot process. To solve this situation cooperation among agents [6] can be introduced. Two kinds of queries can be done to cooperate: event querying and goal querying. If an external event is received by an agent and the agent has no answer (in the form of a goal or reflexive action), the

agent sends a cooperation request to the agents which cooperate with it, represented as a list of cooperation agents, and they will send the applicant agent the goals and actions that they use to attend the event (i.e: their rules which indicate what an event produces). Once received, the agent examines the answers and adds them to the rules of transformation of events in actions and goals. The answers which contain actions that the agent cannot carry out are rejected; actions can be seen as agent's capabilities - if an agent does not have the capability to attend an event, this answer is rejected.

At this point the agent continues its execution but now can turn out that the agent has no plan to reach a certain goal (for example, because it was sent by an agent in response to a cooperation request), in that case the agent sends another cooperation request to get plans to achieve that goal. Received plans are added to the agent's plan repository. Again, plans which contain actions that the agent cannot carry out are rejected.

In order not to interfere with the rest of the agent's behaviour, the cooperation event is not sent to the main process, but it is sent to a cooperation process which is in charge of finding the goal and the appropriate plans and responding with them, if it has any, or with an error event, if not.

The main process must be slightly changed at the point where the received event is processed to deal with the case where there is no response for an event. The change is the following:

1. Receive an event
2. Update beliefs
3. Produce actions and/or goals
4. **If there is no answer at the previous point, a cooperation process is started to ask for help from other agents**
5. **Answers are received and those which contain non-realizable actions are rejected**
6. **If there is no answer, we start again (waiting for new events)**
7. Execute actions.
8. A goal is selected and a plan is launched.

We have to change the `produce_actions_goals` function to deal with this behaviour. The new Erlang code is:

```
produce_actions_goals(State, {Event, Param}) ->
    case state:get_event_action_goal_list(
              State#state.pid_state, Event) of
        {ok, List} ->
            ok = add_actions_goals(State, Param, List),
            true;
        {error, not_found} ->
            case send_cooperation_event(State,
                                        {Event, Param}) of
                {ok, no_answer} ->
                    false;
                {ok, Answer} ->
                    ok = add_actions_goals(State, Param,Answer),
                    true
            end
    end.

send_cooperation_event(State, {Event, Param}) ->
    {ok, CooperationAgents} =
        state:get_cooperation_agents(State#state.pid_state),
    send_cooperation_event_aux(State, CooperationAgents,
                               {Event, Param}).

send_cooperation_event_aux(_, [], _) ->
    {ok, no_answer};
send_cooperation_event_aux(State, [Agent | T],
                           {Event, Param}) ->
    case agente:send_cooperation_event(
                Agent, {event, {Event, Param}}) of
        {ok, no_answer} ->
            send_cooperation_event_aux(
                    State, T, {Event, Param});
        {ok, List} ->
            case lists:any(fun ({action, Value}) ->
                                not state:has_action(
                                      State#state.pid_state,
                                      Value);
                               (_) ->
                                  false
                           end, List) of
                true ->
                    send_cooperation_event_aux(
                            State, T, {Event, Param});
                _ ->
                    {ok, List}
            end
    end.
```

The agent asks its cooperators sequentially and the first valid response is selected. Plan execution process is changed too:

1. Check plan precondition.
2. If precondition fails then the plan is cancelled and a fail event is sent to the main process.
3. Invariant is checked.
4. If the invariant fails, plan is cancelled and a fail event is sent to the main process.
5. Execute next step of the plan.
6. If it is an action, it is executed.
7. If it is a goal, the state of the actual plan is stored and a plan to achieve the goal is found and executed.
8. **If it is a goal and there is no plan, a cooperation request is sent, the answer is received and plans are added to the plan repository (of course, plans that contain actions that cannot be carried out by the agent are rejected)**
9. If execution fails, a fail event is sent to the main process.
10. Continue executing the steps of the plan and checking the invariant until the invariant fails or the plan finished.
11. If plan finished, a finished plan event is sent to the main process.

This cooperation can be seen represented in Fig. 2 Dotted lines in-

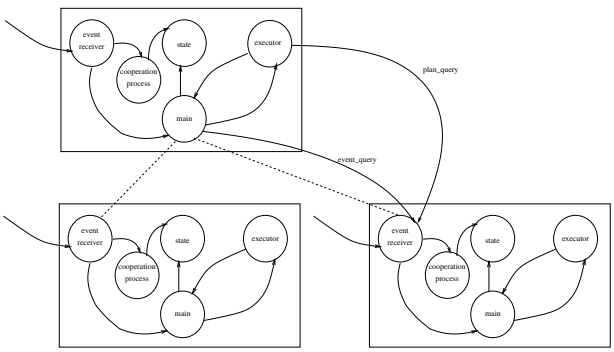

Figure 2. Cooperation among agents

dicate that one process knows the identity of the other process. For implementation purposes what happens is that an agent's main process knows the identity of the agents' events receiver process which cooperate with it. In the figure, a possible communication between two agents is shown. Requests are made from the sender's main process to the receivers' events receiver processes to make event querying, and from the executor process to make plan querying. That kind of cooperation allows the agents to improve their capabilities by means of cooperation with other agents. A new agent can be introduced in the system to "teach" the other agents. Then we need to be able to tell an agent that now it has to cooperate with the new agent in the system, and so agents must support dynamic reconfiguration of its connections. So, reconfiguration in the agents is needed, and that is what we will show in the next section.

A complete classical example of a fire extinguisher robot that can cooperate with other agents can be downloaded at `http://www.lfcia.org/erlang/agents/robots.tgz`

3.2 Reconfiguration

Big distributed systems usually must deal with: (1) hundreds of components that are part of one or more applications, (2) simple applications that are part of bigger systems distributed through a net. The basic goal of remote control is to make a system flexible, highly modifiable at runtime and stable with respect to different sorts of errors. So, it must be possible to do dynamic remote reconfigurations of the different parts of the system, i.e: it should be done while the system is running and possibly as a reaction to an event.

In [7, 8] a system called LIRA is presented which can do configuration changes in its agents. This system does not support well proactivity (i.e: agents in LIRA execute orders of a *Manager*, without any kind of internal reasoning) in addition to coordination issues (see [9] for details). To solve these problems an integration with the DALI system [9] was proposed; DALI is a Prolog-like logic programming language, equipped with reactive and proactive capabilities. DALI is suitable for enhancing Lira agents by implementing a form of intelligence in the agents and adding more communication and cooperation capabilities.

Based on this, solution we propose support for reconfiguration. Two new kinds of agents are introduced in the system: *Managers* and *Reconfiguration Agents*. If a reconfiguration agent is going to be built, it has to be done using an specific *behaviour*. These are the services that must be provided by the agent in our proposal.

- **start(Params)**: It starts the agent with the indicated parameters.
- **stop**: It stops execution of the actual plan and stops receiving events (except *start*)
- **suspend**: It suspends the actual plan execution and stops receiving events (except *resume* and *stop*)
- **resume**: It resumes the execution of the suspended plan
- **shutdown**: It stops the agent execution and kills it

The agent also maintains an internal state (status) with one of the following values: *starting*, *started*, *stopping*, *stopped*, *suspending*, *suspended*, *resuming*, *reconfiguring*; this state can be consulted using the agent's function *status*.

The *Managers* are installed out each node and are capable of starting reconfiguration agents in the same node where they run,

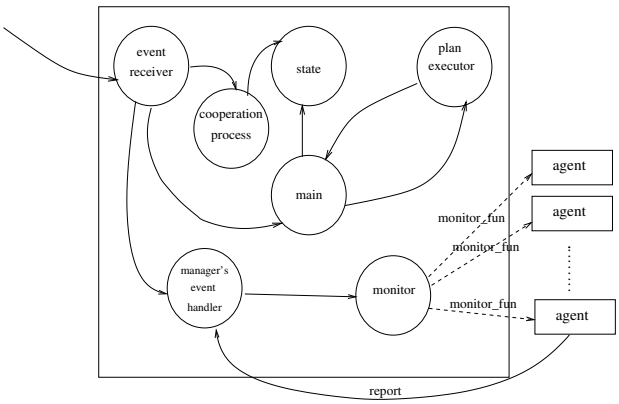

Figure 3. Agent's structure

using the *start* function of the agents' interface. Reconfiguration agents send events to the *Manager* reporting state changes. *Managers* monitor agents' states, some of which can be associated with physical components: if one of them fails, the *Manager* will make changes to the other agents in order to reflect this situation. In fact, a reconfiguration agent that is not a base reconfiguration agent is a manager.

Thus, potentially, all agents can be a *Manager*, so they can deal with a new set of events. These events are:

- An event to add a *Reconfiguration Agent*
- An event to alter a *Reconfiguration Agent*
- Events that are reported from reconfigurations agents that indicate state changes

These events can be represented in Erlang as follows:

```
{add_reconfiguration_agent, AgentID,
 RecoverFun, MonitorFun, OnChangeFun}  % To add agents
{do_in_agent, AgentID, Action} % To alter an agent
{report, AgentID, Value} % A report from an agent
```

The event *add_reconfiguration_agent* receives three functions: A recover function, a monitor function and an on-change function, which represents the behaviour of the *Manager* with respect to the added agent. The on-change function represents the response of the *Manager* to deal with changes in the agent (for example: what to do when they stop or when resumed) - these functions receive a parameter which is one of the values mentioned before. The monitor function checks that everything is correct on the agent and if something goes wrong, the recover function is called.

To add this behaviour to the agents, two new processes are created. These processes are in charge of receiving those events monitoring and altering agents, receiving reports from all of them. This new structure is shown in Fig. 3.

The event manager receives the event from agents (*report*) and the changes to execute over an agent (*do_in_agent*). When receiving an *add_reconfiguration_agent*, it communicates it to the monitor process which is in charge of monitoring agents. The monitor does not receive the on-change function because this functionality is carried out by the manager's event handler.

In this way, dynamic changes can be achieved on agent cooperation, when introducing new agents on the system. *Managers* can be

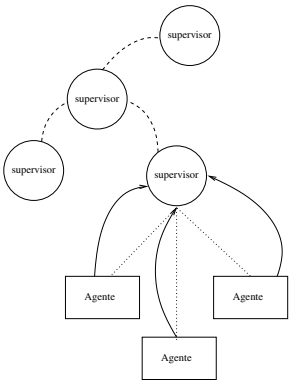

Figure 4. Agents supervision

responsible of changing the configuration of any agent on its node to cooperate with the new agent. As stated in the last section, we can add new knowledge in the existing agents.

In addition, using the agent cooperation infrastructure for plan exchange among agents, changes in agent's configuration not only can be done, but also the agents' behaviour can be explicitly reconfigured to achieve the specified objectives for an agent. To do that, the *Manager* of the node is indicated to change a plan for an agent, and it is in charge of making this change.

3.3 Fault tolerance

With reconfiguration, a first level of fault tolerance has been introduced, but this is not enough to be able to change the agents' configuration faced with the fail of any of them. The desirable behaviour would be able to start the fail agent again with the same state it had before failing.

Erlang is a language designed to support fault tolerance. It allows the linking of two processes so that one process is the supervisor of another process. The supervisor process is reported as soon as the supervised process dies. That allows the creation of supervision trees for fault tolerant system creation. From the point of view of agents at each node, a supervisor process, which controls the state for each agent, is created. This role can be done by a *Manager*, which will have its own supervisor process. If an agent fails, its supervisor process will start it again. Figure 4 shows this scenario.

The following problem arises: The agent must have the same state that it had before the failure. In [10] there is a discussion about volatile and non-volatile knowledge. It should be taken into account what agent's knowledge must be preserved in the face of a fail and what could be generated again. In our agents, the state is represented by the event queues, intentions, goals, beliefs and plans. Events are processed once received, they are sent to the main process where there are not stored. The rest of the information is in the process which stores the state. Of all that information, plans learnt with cooperation can be learnt again, besides the transformation of events in goals and actions. So intentions, goals and beliefs are sent to the supervisor process which will load it again on the agent once it is restarted. Initial plans are loaded on the restart of the agent and learnt plans will be learnt again.

This amount of information which is communicated to the supervisor process does not mean a system overload because this process is located on the same node as the agents which are supervised by it.

4 Future work and conclusions

In this work, Erlang has been presented as a platform to develop agent systems. As processes are a key point in Erlang programming, it is easier to build concurrent systems, and transparency when distributing processes is a desirable quality in building this kind of systems. Hence, it seems useful the use of this language to build agent systems as a concurrent and distributed systems specialisation. The presented approach is attractive to build reactive systems and proactive systems thanks to the incorporation of reflexive actions. A kind of cooperation among agents that permits plans exchange was presented. This kind of cooperation does not allow the agents to interact for doing some collective task, but they cooperate sharing knowledge. Also, the system was provided with a fault-tolerance features using Erlang supervision trees. Higher-order functions and functional values are very useful to model agent behaviour.

As future lines of work, a study of communicating agents using any of the existing ACLs (Agent Communicating Languages) like KQML, FIPA ACL ... is suggested besides doing the agents to cooperate for doing collective task. Adding mobility to the agent system, that could be done by finishing the agent's processes in a node and restarting them in another one, in a similar way as in [3], is also a promising field.

5 References

[1] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[2] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1996.

[3] Johan Arthursson, Jakob Engblom, Ing-Marie Jonsson, Rehan Mirza, Gustaf Naeser, Mikael Olsson, Robert Ottenhag, Dan Sahlin, Maria Schmid, Bertil Spolander, and Elham Zolfonoom. A platform for secure mobile agents. In *Proceedings of The Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology, April 1997.*, pages 109–120, 1997.

[4] K. L. Clark and F. G. McCabe. Go! for multi-threaded deliberative agents. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003. Melbourne, Victoria, July 15th, 2003. Workshop Notes*, pages 17–32, 2003.

[5] Michael Winikoff, Lin Padgham, and James Harland. Simplifying the development of intelligent agents. In *Australian Joint Conference on Artificial Intelligence*, pages 557–568, 2001.

[6] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI Model with Cooperativity. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003. Melbourne, Victoria, July 15th, 2003. Workshop Notes*, pages 146–161, 2003.

[7] M. Castaldi, A. Carzaniga, P. Inverardi, and A.L. Wolf. A light-weight infrastructure for reconfiguring applications. In Bernhard Westfechtel and André van der Hoek, editors, *Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, and Portland, OR, USA,*, volume 2649 of *Lecture Notes in Computer Science*. Springer, May 2003.

[8] S. Porcarelli, M. Castaldi, F. Di Giandomenico, P. Inverardi, and A. Bondavalli. An approach to manage reconfiguration in fault-tolerant distributed systems. In R. De Lamos, C. Gacek, and A. Romanovsky, editors, *Proceedings of the ICSE Workshop on Software Architecture for Dependable Systems*, pages 71–76, May 2003.

[9] M. Castaldi, S. Costantini, S. Gentile, and A. Tocchio. A logic-based infrastructure for reconfiguring applications. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003. Melbourne, Victoria, July 15th, 2003. Workshop Notes*, pages 49–64, 2003.

[10] Paolo Busetta and Kotagiri Ramamohanarao. An architecture for mobile BDI agents. In *Selected Areas in Cryptography*, pages 445–452, 1998.