Erlang as an Implementation Platform for BDI Languages

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 report on our experiences using Erlang to implement a subset of the agent-oriented programming language Jason. The principal existing implementation of Jason is written in Java, but suffers from a number of drawbacks, i.e., has severe limitations concerning the number of agents that can execute in parallel. Basing a Jason implementation on Erlang itself has the potential of improving such aspects of the resulting multi-agent platform.

To evaluate Erlang as a programming language implementation platform the paper describes our experiences in mapping Jason to Erlang, highlighting the positive and negative aspects of Erlang for this task. Moreover, the paper contains a number of benchmarks to evaluate the quantitative aspects of the resulting Jason implementation, especially with respect to support large multi-agent systems.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Performance, Experimentation

Keywords BDI Languages; Erlang Implementation; Translation

1. Introduction

Although not widely known outside the artificial intelligence community, the belief-desire-intention software model (BDI for short) for programming multi-agent systems has a number of interesting features. Basically it provides a higher-level software architecture for separating a number of program concerns:

- *beliefs* are the facts an agent (thinks it) knows
- *desires* are the goals an agent wants to realise, and
- *intentions* represents the strategies the agent has chosen to realise its desire; these are often called *plans*

Moreover, as BDI systems are generally reactive multi-agent systems, *events* (occurrences of actions outside of the control of the agent itself, but which potentially impacts the agent) are key.

Although much of the BDI literature is inspired by the philosophical aspects of artificial intelligence, the concepts above can be reinterpreted to speak about a particular software architecture for a multi-process systems (such as can be implemented in Erlang). We will use that interpretation in this paper.

Erlang'12, September 14, 2012, Copenhagen, Denmark.

As an example then, we consider a classical simple communication protocol using acknowledgement messages (such as the alternating bit protocol or TCP/IP) for transmitting a sequence of bytes, with a sender and received agent (process). What are the beliefs, desires, intentions and events of the agents? Clearly the overall *desire* of both the sender and receiver agents is that the sequence of bytes gets correctly transferred and received at the receiver agent. *Beliefs* concern basic facts, as well as more complex facts concerning protocol negotiation. For instance, a basic belief of the sender is that it has sent some subsequence of bytes on the communication channel. A more complex belief is that the sender "knows" that the receiver agent has correctly received a sent subsequence of bytes; this knowledge comes from observing the *event* that an "acknowledgment" message was received from the receiver agent. Similarly a receiver agent may possess the more complex belief that it "knows" that the sender agent knows that the receiver agent has correctly received a subsequence of bytes (usually this knowledge comes from having received a later subsequence of bytes as an event). The *intentions*, or *plans*, describe the concrete protocol, i.e., for an agent, when a new event occurs, and given a certain set of beliefs, what is the next set of actions of an agent. An action may be a communication or that the agent updates its beliefs. As an example, the sender agent may send an new sequence of bytes upon receiving an acknowledgement message (event) from the receiver agent, and also update its beliefs to reflect the fact that it knows that the receiver agent has correctly received the previous part of the message.

Compared to a more traditional way of implementing a communication protocol, here the main the difference is that the knowledge of the agents (processes) is structured and made explicit in a declarative fashion.

There are numerous different BDI programming language systems around; some of the more popular ones are Jason [2], Goal [5] and 2APL [3]. As a core part, most of the programming language systems use a prolog inference engine to resolve basic queries regarding beliefs, to store beliefs, and to select plans. Although all these language systems do provide support for multi-agent systems, i.e., adequate communication facilities, it is fair to say that the major implementation effort for systems like e.g. Goal has been on improving basic inference speed by utilising a high-performing prolog implementation. Rather less attention has been spent on improving multi-agent aspects, resulting in multi-agent system implementations that do not scale to more than at most a few thousand concurrent agents.

Finding the BDI software architecture interesting, while being disappointed in the actual quality of the existing implementations with regards to supporting large agent based systems, naturally we considered how to integrate the high-performing Erlang runtime system in such a BDI platform. One possibility would have been to add support for beliefs and plans to Erlang itself, possibly as behaviours. A similar approach is followed by the agent implementation platform eXAT [9]. Instead we chose to implement a particular

BDI *language*, Jason, using Erlang. The main reason for implementing Jason was to be able to establish a dialogue with mainstream BDI researchers, who we believe, would not have been particularly interested in Erlang with some BDI sugar added.

The main contribution of this paper is as a case study in using Erlang as a platform to implement, using a mix of compilation and interpretation techniques, another programming language (Jason). It turned out be rather straightforward to use Erlang to realise an implementation of the chosen subset of Jason, as the languages have a lot of features in common: they are syntactically similar, both having roots in Prolog (uppercase variables, atoms, single assignment variables, etc.), and they are both based on an Actor communication model. Moreover, although not a focus of this paper, fault tolerance is recognised as a problem for agent-oriented platforms, and it is hoped that the failure detector primitives of Erlang (links and monitors) can be integrated into the resulting multi-agent platform.

A prototype of the implementation of Jason in Erlang, named *eJason* is available for downloed at

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

In Sect. 2 we further describe the BDI model and Jason in particular, while Sect. 3 describes how our Jason implementation was realised using Erlang. This Jason implementation is benchmarked in Sect. 4, while Sect. 5 describes items for further work.

2. Belief-Desire-Intention Model

Jason is both a programming language which is an extension of AgentSpeak [7], and an interpreter of this programing language in Java. It is based on the Belief-Desire-Intention architecture[8, 12] which is central in the development of multiagent systems. This approach allows the implementation of rational agents by the definition of their know-how, i.e. how each agent must act in order to achieve its goals, instead of a more classical reactive model.

Following a BDI architecture, the constructs of the Jason programming language can be separated into three main categories: beliefs, goals and plans. In the remainder of this section we describe each of them via an example.

2.1 A simple example: Teacher and Pupil

Consider a multiagent system composed by two agents. The first agent is a pupil agent whose only purpose is to count from some initial number `I` to a maximal number `M`. Given a certain integer number, the pupil agent cannot compute the next number but needs to ask a teacher agent the value of the following number by sending a message to the teacher agent. The teacher agent has no initial goal, but every time that it receives an integer number `X` from a pupil agent, it computes the following integer number `Y = X+1` and sends it back to the pupil. Every time that the pupil receives a new integer number from the teacher, it checks whether it has reached the maximum integer `M` or not. In the first case, it just prints the message "Terminated Count" in the standard output. Otherwise, it asks the teacher the following number again.

This example represents a sample multiagent system where the behaviour of the agents is simple but that requires an intensive communication between the agents included. The Jason source code for the pupil and teacher agents is presented in Figures 1 and 2, respectively.

2.2 Jason Beliefs

Agents in Jason have beliefs which somehow represent the information that the agent has currently been able to obtain about its environment, including other agents. The set of all beliefs of an agent in Jason is contained in a structure named *belief base*. This

```
init_count(0).
max_count(100).

!startcount.

+!startcount : init_count(X) <-
         +actual_count(X).

+actual_count(X) :  max_count(Y)& X < Y <-
        -actual_count(X);
        .send(teacher,tell,actual_count(X)).

+actual_count(X): max_count(Y)& X >= Y <-
            .print("Terminated count").
```

Figure 1. Jason code for the pupil agent

```
next(X,Y) :-
     Y = X +1.


+actual_count(Count)[source(Pupil)]:true <-
    -actual_count(Count)[source(Pupil)];
    ?next(Count,Next);
    .send(Pupil,tell,actual_count(Next)).
```

Figure 2. Jason code for the teacher agent

belief base is updated dynamically during the life of the agent due to different causes described in Section 2.5.

The belief base of an agent is composed by a set of ground predicates (i.e. predicates without unbound variables) referred as *beliefs* and a set of *rules* that allow the inference of new knowledge from the information already possessed. The knowledge base can be accessed in order to determine if a plan context can be matched or to match certain test goal, as described below.

In Fig. 1, the belief base of the pupil consists of the following beliefs:

```
init_count(0).
max_count(100).
```

representing that the agent believes the initial value to be zero, and that the agent believes that it has to count up to 100.

The teacher agent in Fig. 2 does not have any initial beliefs in its belief base but it does have the following rule:

```
next(X,Y) :-
     Y = X +1.
```

This rule is used to calculate the successor of a given number.

2.3 Jason Goals

Goals express the properties of the states of the world that the agent wishes to bring about. In Jason, there are two types of goals: achievement goals and test goals. Achievement goals are denoted by the '!' operator. So, for example, the initial goal of the pupil agent in Fig. 1, `!startcount`, means that the pupil agent has the goal of achieving a certain state of affairs in which the agent will believe it starts counting.

Test goals are normally used simply to retrieve information that is available in the agent's belief base. They start with the '?' operator.

2.4 Jason Plans

Plans represent the know-how of a program, i. e., the knowledge about how to do things, which is used to reach the agents goals. Besides, in Jason, plans are also used to characterise responses to events.

Plans in Jason have three distinct parts: the triggering event, the context, and the body. The three plan parts are syntactically separated by ':' and '<-'

Triggering events are related to two types of changes in the agent's mental attitude: changes in beliefs (which can refer to the information agents have about their environment or other agents) and changes in the agent's goals.

For example, the triggering event in the first of the three plans of the pupil agent in Fig. 1 is the pupil initial achievement goal `!startcount`.

```
+!startcount : init_count(X) <-
         +actual_count(X).
```

The context of a plan is used for checking the current situation so as to determine whether a particular plan, among the various alternative ones, is likely to succeed in handling the event (e.g. achieving a goal), given the latest information the agent has about its environment. A plan that has a context which evaluates as true given the agent's current beliefs is said to be applicable at that moment in time, and is a candidate for execution. In the previous example of a plan of the pupil agent, the context `init_count(X)` evaluates to true initially since `init_count(0)` belongs to the pupil's initial belief base.

The body of a plan is a sequence of formulae determining a course of action for the agent to take when an event that matches the plan's triggering event has happened and the context of the plan is true in accordance with the agent's beliefs (and the plan is chosen for execution). For instance, in the previous example of a plan of the pupil agent, the body of the plan is to add the belief `actual_count(0)` to the pupil's belief base.

The second plan of the the pupil agent in Fig. 1 is the following:

```
+actual_count(X) :  max_count(Y)& X < Y <-
        -actual_count(X);
        .send(teacher,tell,actual_count(X)).
```

Here, the context is used to check that the pupil has not reached the maximum number. In that case, the belief `actual_count(0)` is removed from the pupil's belief base and the pupil asks the teacher the successor of zero.

Lastly, the third plan of the the pupil agent in Fig. 1 is the following:

```
+actual_count(X): max_count(Y)& X >= Y <-
            .print("Terminated count").
```

If the pupil has reached the maximum number, the pupil prints a message to indicate that it has finished counting.

The teacher agent in Fig. 2 has only one plan:

```
+actual_count(Count)[source(Pupil)]:true <-
    -actual_count(Count)[source(Pupil)];
    ?next(Count,Next);
    .send(Pupil,tell,actual_count(Next)).
```

Upon the triggering event of adding the belief `actual_count` with initial value zero to the teacher belief base, the belief will be removed from the teacher belief base, the successor value of the number, zero in the initial case, will be retrieved and the new value will be communicated to the pupil.

2.5 Jason Reasoning Cycle

The semantics of Jason are given by its reasoning cycle. This reasoning cycle is depicted in Figure 3. The main features of this cycle are the following:

- An agent obtains information both from its environment (perception) and from the messages received from other agents. This information may update the belief base and generate new events that must be handled by the agent.
- An agent may modify its environment by executing some action or may interact with other agents via asynchronous message passing.
- In every iteration, a single event is chosen to be processed, by the event selection function. The set of relevant plans for that event are those plans in the plan library whose trigger matches it. For each relevant plan identified, the belief base is queried to determine whether its context is met. The set of applicable plans are those relevant plans whose context holds.
- An option selection function chooses one applicable plan from a list of these. The plan chosen is added to the set of intentions of the agent.
- Each intention is composed by a set of partially instantiated plans, i.e. a stack of formulae (appearing in the body of a plan) to be executed.
- An intention selection function selects one intention from the set of intentions of the agent. The formula on top of this intention is executed and the remaining stack is returned to the set of intentions of the agent.

A detailed explanation of the reasoning cycle of Jason is beyond the scope of this work. It is provided in [2].

2.6 Prolog as Inference Engine

As stated above, a Jason agent possesses a mechanism to infer knowledge from its set of beliefs, the *rules*, and to determine whether the *context of a plan* holds. Both these elements are represented in Jason using a Prolog-like predicate syntax. Therefore, Jason utilises a Prolog inference engine to determine if the belief base implies certain predicate, which can be part of a rule or a plan context.

3. Implementation

In this section we present the two main building blocks of the Erlang implementation of Jason. The first one is a **translator**, built using the third-party software Yecc, which is an Erlang implementation of a LALR(1) parser similar to the well-known parser Yacc [6], that generates Erlang source code from the Jason source for the agents in the system. Thus this translator deals with the *syntactical facet* of the implementation. The second is a set of Erlang modules that implement the reasoning cycle of Jason. As stated in Sect. 2.5, this reasoning cycle provides the semantics of Jason. Therefore, the Erlang code for the **reasoning cycle** represents the *semantic facet* of the implementation. In the remainder of the section we provide more details for these two components.

3.1 eJason Translator

The eJason translator allows the automatic generation of the Erlang code corresponding to some syntactically valid Jason code, i.e. its translation. As depicted before, for each "[filename].asl" file, corresponding to a different Jason agent, a new "[filename].erl" is produced. The latter ones contain a series of functions that are invoked either to setup the initial state of the agent or to execute some part of a plan. As a matter of example, the excerpt of Erlang

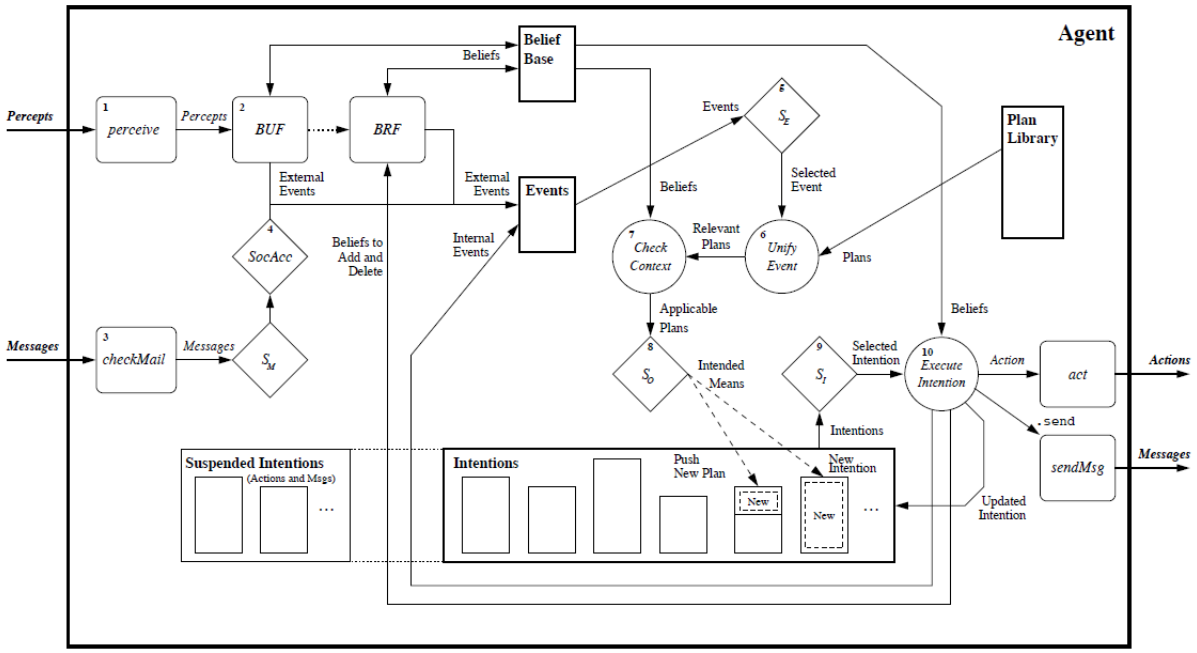

Figure 3. Reasoning Cycle of Jason.

code in Figure 4 corresponds to the Jason code for the pupil already presented in Figure 1. For the sake of readability of the code reported, next we describe how the different components of Jason are represented in Erlang.

3.1.1 Variables

In our implementation, we consider as an eJason variable every variable in Jason and any atom appearing in the body of a Jason predicate. In the Jason subset considered, variables appear only in Jason plans and rules. In order to evaluate these variables, we use a *valuation* structure. Concretely, every valuation is represented by an Erlang tuple where the values are associated with the distinct variables ordered according to the inverse order in which these variables first occur in the plan or the rule. For instance, the variables appearing in the second plan of a pupil agent are `{tell,teacher,Y,X}` and a valid valuation for them (actually, the one corresponding to the initial state of the agent) would be `{tell,teacher,'_',0}`, thus binding `X` to 0 and leaving `Y` unbound.

3.1.2 Predicates

Predicates appear in every Jason structure. Every predicate is composed by three different elements: a name, a set of parameters (which are either atoms, variables or other predicates) and a list of annotations (which are themselves predicates too). Each Jason predicate is represented in eJason by a tuple with three elements. The first element is an atom that corresponds to the name of the predicate. The second element is a tuple that contains the parameters, which can be empty in the case of predicates that take no parameters (as is the case for atoms). The third and last element is a possibly empty list containing all the annotations of the predicate, all of which are predicates, hence represented as tuples of three elements. For instance, the Jason predicates `startcount` and `actual_count(Count)[source(Pupil)]` are represented in eJason (respectively) as `{startcount,{},[]}` and `{actual_count,{Count},[{source,{Pupil},[]}]}`.

3.1.3 Beliefs

Every belief is a predicate (notice that atoms are considered as predicates that take no parameters), so each of them is represented as a tuple with three elements as described above.

3.1.4 Goals

Jason allows two kinds of goals: achievement and test goals. They are represented in eJason using a tuple with two elements. The first element is either the atom `add_achievement_goal` or the atom `add_test_goal`. The second element is the representation of the predicate that represents the goal. For instance, the achievement goal `!startcount` is represented as `{add_achievement_goal,{startcount,{},[]}}`.

3.1.5 Rules.

Each rule in Jason is represented as an Erlang function. This function, when provided with the proper number of input parameters, accesses the belief, if necessary, and returns the list of all the terms that both satisfy the rule and match the input pattern.

3.1.6 Plans

The plans in Jason are the structure that posed the biggest challenge to the translation. Nevertheless, they can be implemented in Erlang using the following Erlang functions:

- One function to implement the trigger of the plan. This function will either return a variable valuation (possibly with unbound variables) if the trigger succeeds or the atom `false`, if the plan is not applicable.
- One function to implement the plan context. This function will return a list of valuations that comply with the plan context. If the list is empty, the plan is not applicable.

- As many functions as formulae are there in the plan body. Each of these functions executes the semantics of a different formula, e.g. modifying the belief base by adding or removing beliefs and generating new achievement or test goals.
- A function that is executed when the plan has been executed entirely, i.e. when all the formulae in its body have been executed. This last function returns the valuation for the variables appearing in the plan trigger, if any.

For convenience, each plan is represented using a record `plan` whose elements are its trigger function, its context function and a list with the functions for the formulae in the body. This list is ordered in order of appearance and the last one is the function to be executed after a complete execution of the plan, as mentioned above.

For instance, the second plan in Figure 1 can be represented as

```
#plan{trigger=fun pupil:actual_count_2_trigger/1,
    context=fun pupil:actual_count_2_context/2,
    body=[fun actual_count_2_body_formula_1/2,
          fun actual_count_2_body_formula_2/2,
          fun actual_count_2_body_last_formula/1]}
```

3.1.7 Inference Engine in eJason

As mentioned in Section 2.6, Jason relies on a Prolog inference engine to resolve whether the formulae in a plan context hold, using the knowledge in the belief base. On earlier stages of the implementation, we used the third party software ERESYE [10] (ERlang Expert SYstem Engine) to implement both the belief base of each agent and its inference engine. ERESYE is a library to write expert systems and rule processing engines using the Erlang programming language. It allows to create multiple engines, each one with its own facts and rules to be processed. In a later stage, we decided to implement our own belief base and Erlang inference engine. This new implementation provides a reduced functionality (compared to ERESYE) that, nonetheless, fits better to our purpose.

The new inference engine requires the generation of an Erlang function for each rule and plan context. As stated above, that function returns a list of valuations that comply with the plan context or the body of a rule. If the list is empty, neither the context is met nor a rule can be applied to infer new knowledge.

3.1.8 Starting an Agent in eJason

The behaviour of each agent in eJason is executed by a different Erlang process. Therefore, starting an agent implies spawning a new Erlang process and executing a set of functions that compute its initial state. These functions are:

- **start(Num):** this function is executed every time that a new agent is started. Its parameter `Num` is used for convenience and indicates the spawning order assigned to the agent. It is used when more than one instance of the same agent is spawned. For instance, to spawn `N` pupil agents, eJason will execute `spawn(pupil,start,1)`, `spawn(pupil,start,2)` ... `spawn(pupil,start,N)`. The function `start/1` first registers the new process with the name of the agent followed by its spawning order, excepting when it is 1, i.e. for the example above the new `N` processes would be registered as `pupil`, `pupil'_2` ... `pupil'_N`. Then, it invokes the proper functions to compute the initial state of the agent. These functions are described below.
- **addInitialBeliefs(Agent):** this function modifies the agent state received as a parameter in order to add the proper initial beliefs to the belief base of the agent. In figure 4 the initial beliefs added are `init_count(0)` and `max_count(100)`.
- **addInitialGoals(Agent):** analogously to the previous function, it modifies the state of the agent received as parameter to include the events corresponding to its initial goals. Again in Figure 4, the initial goal added corresponds to `!startcount`.
- **addPlans(Agent):** this function adds its proper plan base to the agent received as parameter. Recall that each plan is represented by a `plan` Erlang record. Then, in Figure 4, four plans are added: one plan triggered by the achievement goal `!startcount`, two plans to handle the addition of a belief `+actual_count(Count)` and one plan to resolve every test goal.
- **reasoningCycle:reasoningCycle(Agent):** this functions implements the reasoning cycle of the Jason agents. It is executed in a loop during by each eJason agent during all its lifetime. It receives as parameter the current state of the agent.

3.2 eJason Reasoning Cycle

The Jason reasoning cycle [1] must of course be represented in eJason. We implement the reasoning cycle using an Erlang function `reasoningCycle` with a single parameter, an Erlang record named *agentRationale*, which represents the current state of the agent. The elements of this record are: an atom that specifies the name of the agent, a list that stores the events that have not yet been processed, the list of executable intentions for the agent, the list of executable plans, a list of the terms that compose the agent belief base, and three elements (`selectEvent`,`selectPlan` and `selectIntention`) bound to Erlang functions implementing event, plan and intention selection for that particular agent (in this manner each agent can tailor its selection functions; appropriate defaults are provided).

Below a sketch of the `reasoningCycle` function is depicted, providing further details on how eJason implements the reasoning cycle of Jason agents:

This record is updated during the execution of each reasoning cycle:

(1) At the beginning of each reasoning cycle, the agent checks its mailbox and processes its incoming messages, adding new events.

(2) The event selection function included in the *agentRationale* record is applied to the list of events also included in the same record. The result of the function evaluation is an Erlang record of type *event*. This record represents the unique event that will be processed during the current reasoning cycle.

(3) The function trigger of every plan is applied to the body of the selected event. For every distinct valuation returned by a trigger function, a new plan is added to the list of *relevant plans*. Each relevant plan is represented by a *plan* record along with a valuation for the parameter variables.

(4) Next, the context function of each relevant plan is evaluated. The result of each function application is either an extended valuation, possibly binding additional variables, or the failure to compute a valuation that is consistent with both the trigger and the context. For each remaining valuation, a new plan is added to the set of applicable plans. Each applicable plan is represented by a set of variable bindings along with a *plan* record.

(5) The plan selection function is applied to the list of applicable plans. The result obtained is an applicable plan that represents the new intended means to be added to the list of intentions.

(6) The intention selection function is applied to the list of executable intentions. It selects the intention that will be executed in the current reasoning cycle. Note that, as specified by the Ja-

```
-module(pupil).
-compile(export_all).
-include("macros.hrl").

-define(Name,pupil).
-define(Internal,pupil).
-define(Environment,pupil).
-define(FunNames,[]).

start()->
  start(1).

start(Num)->
  AgName= utils:register_agent(Num,self(),pupil),
  Agent0 = reasoningCycle:start(AgName,[],[],
           beliefbase:start()),
  Agent1 = addInitialBeliefs(Agent0),
  Agent2 = addInitialGoals(Agent1),
  Agent3 = addPlans(Agent2),
  reasoningCycle:reasoningCycle(Agent3).

%% Function to includeInitial beliefs.

addInitialBeliefs(Agent = #agentRationale{})->
  reasoningCycle:applyChanges(Agent,[
    {add_belief,{init_count,{0},[]}},
    {add_belief,{max_count,{100},[]}}]).

%% Function to include initial goals.

addInitialGoals(Agent = #agentRationale{})->
  InitGoalList=[{addEvent,#event{type=external,
    body={add_achievement_goal,
          {startcount,{},[]}}}}],
  reasoningCycle:applyChanges(Agent,InitGoalList).

addPlans(Agent = #agentRationale{})->
Agent#agentRationale{plans = [

#plan{trigger=fun ?Name:actual_count_3_trigger/1 ,
  body= [{fun actual_count_3_body_formula_1/2,
            [{3,1}]},
    fun actual_count_3_body_last_formula/1],
    context=fun ?Name:actual_count_3_context/2},

#plan{trigger=fun ?Name:actual_count_2_trigger/1 ,
  body= [{fun actual_count_2_body_formula_1/2,
           [{4,1}]},
{fun actual_count_2_body_formula_2/2,
           [{1,1},{2,2},{4,3}]},
    fun actual_count_2_body_last_formula/1],
  context=fun ?Name:actual_count_2_context/2},

#plan{trigger=fun ?Name:startcount_1_trigger/1 ,
  body= [{fun startcount_1_body_formula_1/2,
           [{1,1}]},
    fun startcount_1_body_last_formula/1],
  context=fun ?Name:startcount_1_context/2},

#plan{trigger=fun
  ?Name:ejason_standard_test_goal_handler_trigger/1,
  body=[fun
    ?Name:ejason_standard_test_goal_handler_body/2],
  context=fun
    ?Name:ejason_standard_test_goal_handler_context/2}]}.

startcount_1_trigger({add_achievement_goal,startcount})->
  InitValuation = list_to_tuple(lists:duplicate(2,'_')),
  ResVal = utils:updateValuation([InitValuation],
     [{startcount}],[{2,1}]),
     case ResVal of
       []-> false;
       _-> {true,ResVal}
     end;
startcount_1_trigger(_A)->
   false.

startcount_1_context(BBID,InitVal)->
  Val0 = case InitVal of
           A when is_list(A) -> InitVal;
           A when is_tuple(A) -> [A]
         end,
  Par1 =utils:makeParams(Val0,[1]),
  Res1 = utils:query_bb(?Name,BBID,{init_count,'_','_'},
  Par1,?FunNames),
  Val1 = utils:updateValuation(Val0,Res1,[{1,1}]),
  Val1.

startcount_1_body_last_formula(Valuation)->
[{finished,utils:makeParams([Valuation],[2])}].

startcount_1_body_formula_1(BBID,Valuation)->
  VarNames = ['X',startcount],
  Element = utils:unify_vars({actual_count,{'X'},[]},
    Valuation,VarNames),
  NewEvent = utils:add_belief(Element),
  Res = [NewEvent],
  Res.
```

Figure 4. Excerpt of Erlang code for agent Pupil

```
reasoningCycle(OldAgent) ->
    Agent = check_mailbox(OldAgent),                          (1)
    #agentRationale
       {events = Events,
        belief_base = BB,
        agentName = AgentName,
        plans = Plans,
        intentions = Intentions,
        selectEvent = SelectEvent,
        selectPlan = SelectPlan,
        selectIntention = SelectIntention} = Agent,

    {Event,NotChosenEvents} = SelectEvent(Events),            (2)
    IntendedMeans =
      case Event of
        [] -> [];  %% No events to process
        _ ->
         RelevantPlans = findRelevantPlans(Event,Plans),      (3)
         ApplicablePlans = unifyContext(BB, RelevantPlans),   (4)
         SelectPlan(ApplicablePlans)                          (5)
      end,
    AllIntentions = % The new list of intentions is computed
        processIntendedMeans(Event,Intentions,IntendedMeans),
    case SelectIntention(AllIntentions) of                    (6)
       {Intention,NotChosenIntentions} ->
          Result = executeIntention(BB,Intention),            (7)
          NewAgent =                                          (8)
            applyChanges
            (Agent#agentRationale
                {events = NotChosenEvents,
                 intentions = NotChosenIntentions},
                 Result),
        reasoningCycle(NewAgent);                             (9)
    end.
```

Figure 5. Erlang code for the Jason Reasoning Cycle (annotated)

son formal semantics, this intention may not necessarily be the intention that contains the intended means for the event processed at the beginning of the reasoning cycle.

(7) The first remaining formula of the plan that is at the head of the chosen intention is evaluated. The result of evaluating a function may generate new internal or external events, e.g. by adding a new belief to the belief base.

(8) The new events generated are added to the list of events stored in the *agentRationale* record representing the state of affairs of the agent. If the formula evaluated was the last one appearing in a plan body, the process implementing the plan body terminates. If, moreover, the plan that finished was the last remaining plan in the corresponding intention, the intention itself is removed from the list of executable intentions.

(9) Finally a new reasoning cycle is started by repeating steps 1-9 with the new updated *agentRationale* record.

4. Evaluation

In this section we provide an evaluation, based on our experience, of the suitability of Erlang as platform to implement a BDI language. This evaluation is based on the two aspects that we consider of most relevance: the difficulty of the implementation and the performance comparison with an already existing implementation in a different programming language.

4.1 Difficulty of implementation

Due to the high similarity in the syntax of both programming languages, the implementation of the syntactic elements of Jason in Erlang was quite straightforward. The effort invested in this aspect was of approximately 0.25 Man/Months.

Nevertheless, the implementation of the semantics of Jason was not trivial. For the subset considered, the design and implementation phases required 2.25 Man/Months. The most challenging task was the implementation of the semantics for plans, specially their context. These constructs require the implementation of some mechanism to determine whether the context of a plan can be inferred from the knowledge of the agent (i.e. its beliefs and rules). This mechanism should take into account the different possible valuations of the variables for each of the statements in a plan context, as well as determining which of those were valid. Also, the implementation of a plan body was quite challenging, as it requires the realisation of changes of context between different iterations of the reasoning cycle in order to allow the parallelization of the execution of the formulae in the intentions of the agents.

On the other hand, the implementation of the features related to an actor-based execution model (e.g. asynchronous communication, concurrent execution of agents or agent registration) were also straightforward. They are provided inherently by the Erlang Runtime System through the use of the proper constructs.

4.2 Performance

Another relevant factor to consider is the comparison of the performance of the different implementations of Jason. As the similarities between the execution model of Jason and Erlang are much higher than those between Jason and Java, we expected our implementation to allow a significant improvement in the performance of the Jason systems executed under eJason.

In order to check whether our expectations were met, we built several multiagent systems and compared the performance of their execution using eJason and the Java-based implementation of Jason. Table 1 shows the execution times for the multiagent system provided as example in Section 2.1. The figures in Tables 2 and 3 are originally included in [4] and are reported again here for the sake of self-containment of the present document.

As the figures show, considering the execution times of each platforms for each system we can state that the improvement in the performance is of great significance. Some systems require several seconds or minutes, in the best case, to be executed by the Java implementation while they can be executed in a matter of milliseconds by eJason. Moreover, some experiments, the ones composed by a relatively big number of agents, could not be run under Java as some execution exceptions were raised. We label these latter cases in the tables as "not measurable".

Number of Pupil Agents	Jason execution time (magnitude)	eJason execution time (milliseconds)
10	milliseconds	< 1
100	seconds	764
1000	not measurable	4646
10000	not measurable	50000
100000	not measurable	722535

Table 1. Execution times for the pupil and teacher multiagent system

Number of Agents	Jason execution time (magnitude)	eJason execution time (milliseconds)
10	milliseconds	2
100	milliseconds	46
1000	seconds	181
10000	minutes	1916
100000	not measurable	18674
500000	not measurable	97086
800000	not measurable	165522

Table 2. Execution times for the counter multiagent system

Number of Agents	Jason execution time (magnitude)	eJason execution time (milliseconds)
10	milliseconds	1
100	milliseconds	15
1000	seconds	143
10000	minutes	1550
100000	not measurable	154415
300000	not measurable	484371

Table 3. Execution times for the greetings multiagent system

5. Conclusion and Future Work

Early benchmark figures indicate that eJason does scale well with regards to both the number of agents, as with regards to the number of messages sent. The difference compared to the standard Java based Jason implementation is quite significant. Moreover the Erlang implementation of Jason is rather clean and compact, and the manner in which a Jason implementation is supposed to be parametric with regards to a number of key selection functions (e.g., selecting one plan among a set of executable plans) is elegantly implementable in Erlang.

However, it is to be expected that benchmarks that stress the logical inference part of Jason, which is quite similar to Prolog, and normally implemented using a Prolog interpreter, would favour the standard Jason implementation. Currently our implementation of this inference engine is written in Erlang, and needs to be optimized. An option would be to use one of the Prolog interpreters written in Erlang; Robert Virdings' erlog [11] is probably the most well known implementation. Even so, we can expect the performance of an Erlang based Prolog interpreter to perform significantly worse than using a standard Prolog interpreter, as e.g. a normal Prolog system uses randomly accessed stacks heavily, a data structure which is difficult to implement efficiently in pure Erlang.

Further work includes finishing the eJason implementation, which currently for instance lacks support for crucial belief annotations (what is the source of a belief). Once this implementation is finished, we plan to evaluate its perfomances using some widely accepted benchmark for multiagent systems, if it exists. Then we expect to experiment with the BDI software architecture itself, to determine whether as expected more classical distributed system algorithms and programs can benefit from being rephrased using this architectural design pattern. In another strand of research, we expect to benefit from the design and implementation of Erlang in order to augment the Jason language with better support for concurrent and distributed computations. For instance, a recognised problem with Jason is the poor support for fault tolerance. We suspect that the Erlang experience of mapping low-level failure detection primitives to fault handling in higher-level design pattern such as e.g. the generic server behaviour is the right approach for Jason too.

Acknowledgments

This work has been partially supported by the following projects: DESAFIOS10 (TIN2009-14599-C03-03) and PROMETIDOS (P2009/TIC-1465).

References

[1] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12:2006, 2006.

[2] 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.

[3] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008. ISSN 1387-2532. doi: 10.1007/s10458-008-9036-y. URL `http://dx.doi.org/10.1007/s10458-008-9036-y`.

[4] 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.

[5] K. Hindriks, F. de Boer, W. Van Der Hoek, and J. Meyer. Agent programming with declarative goals. *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 248–257, 2001.

[6] S. Johnson and i. Bell Telephone Laboratories. *Yacc: Yet another compiler-compiler*. Bell Laboratories, 1975.

[7] 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.

[8] 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.

[9] A. D. Stefano and C. Santoro. eXAT: an experimental tool for programming multi-agent systems in Erlang. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*, 2003.

[10] A. D. Stefano, F. Gangemi, and C. Santoro. ERESYE: artificial intelligence in erlang programs. In K. F. Sagonas and J. Armstrong, editors, *Erlang Workshop*, pages 62–71. ACM, 2005. ISBN 1-59593-066-3.

[11] R. Virding. web page. `https://github.com/rvirding/erlog`, April 2012.

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