# eJason: an implementation of Jason in Erlang*

Álvaro Fernández Díaz, Clara Benac Earle, and Lars-Åke Fredlund

Babel Group. Universidad Politécnica de Madrid, Spain
{avalor,cbenac,fred}@babel.ls.fi.upm.es

**Abstract.** In this paper we describe eJason, a prototype implementation of Jason, the well-known agent-oriented programming language, in Erlang, a concurrent functional programming language. The reason for choosing Erlang as the implementation vehicle is the surprising number of similarities between Jason and Erlang, e.g., both have their syntactical roots in logic programming, and share an actor-based process and communication model. Moreover, the Erlang runtime system implements lightweight processes and fast message passing between processes. Thus, by mapping Jason agents and agent-to-agent communication onto Erlang processes and Erlang process-to-process communication, we can create a very high-performing Jason implementation, potentially capable of supporting up to a hundred thousand concurrent actors. In this paper we describe in detail the implementation of Jason in Erlang, and provide early feedback on the performance of the implementation.

## 1 Introduction

Among the different agent-oriented programming languages, AgentSpeak [13] is one of the most popular ones. It is based on the BDI architecture [14, 17], which is central in the development of multiagent systems. AgentSpeak 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. AgentSpeak has been extended into a programming language called Jason [7, 9]. Jason refers to both the AgentSpeak language extension and the related interpreter that allows its execution in Java. Thus, Jason is an implementation of AgentSpeak that allows the construction of multiagent systems that can be organized in agent infrastructures distributed in several hosts. It allows the interfacing to the JADE Framework [5, 6], thus generating multiagent systems fully compliant to FIPA [1, 12] specifications. To effortlessly distribute the agent infrastructure over a network, the use of the SACI [11, 2] middleware is suggested. Jason has been designed to address the desirable properties of rational agents identified in [16]: autonomy, proactiveness, reactiveness and social ability. In the rest of the paper we assume that the reader is familiar with Jason [9].

A significant new trend in processor architecture has been evident for a few years. No longer is the clock speed of CPUs increasing at an impressive rate,

* This work has been partially supported by the following projects: DESAFIOS10 (TIN2009-14599-C03-00) and PROMETIDOS (P2009/TIC-1465).

rather we have started to see a race to supply more processor elements in mainstream multi-core CPU architectures coming from Intel and AMD. Initially, the software industry has been slow in reacting to this fundamental hardware change, but today, utilising multiple cores is the only way to improve software system performance. With traditional programming languages (such as Java, C, C++, etc.) writing bug-free concurrent code is hard, and the complexity grows quickly with the number of parallel tasks. As a result, alternative languages, with less error-prone concurrency primitives, are attracting more attention.

Following this trend, the Erlang programming language [3, 10] is gaining momentum. The usage has increased, and among the users are large organisations like Facebook, Amazon, Yahoo!, T-Mobile, Motorola, and Ericsson. The most prominent reasons for the increased popularity of Erlang are lightweight concurrency based on the actor model, the powerful handling of fault tolerance, the transparent distribution mechanisms, the generic OTP design patterns, and the fact that the language has functional programming roots leading to a small, clean code base.

In this paper we report on our experience translating the Jason programming language to Erlang. The similarities between Jason and Erlang – both are inspired by Prolog, both support asynchronous communication among computational independent entities (agents/processes) – make the translation rather straightforward. By implementing Jason in Erlang we offer the possibility to Erlang programmers of using an agent-oriented programming language like Jason integrated in Erlang. To Jason programmers, the approach gives them the possibility of executing their code in the Erlang runtime system, which is particularly appropriate for running robust multiagent systems with a large number of concurrent and distributed agents.

Moreover, as the syntax of Erlang is inspired by Prolog [1], e.g., having atoms beginning with a lowercase letter, and single-assignment variables beginning with an uppercase letter, etc., we hope to reduce the conceptual gap for a Jason programmer interested in modifying the Jason meta-level (e.g., changing the selector functions, and implementing actions) by adopting Erlang, compared to having to use Java. Perhaps even more interesting is the potential for introducing Erlang programmers to the world of BDI programming through this new Jason implementation. This is a group of programmers already used to thinking of programming systems composed of independent communicating agents (or in the terminology of Erlang, processes), and superficially familiar with the syntax of Jason. To us it appears that the conceptual gap between programming agents in Jason, and functions and processes in Erlang, is smaller than for many other programming languages (Java).

A prototype of the implementation of Jason in Erlang is available at

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

The rest of the paper is organized as follows. Before explaining the translation, the main characteristics of Erlang are briefly described in Sect. 2. Then,

[1] Not surprisingly, as the first implementation of Erlang was written in NU Prolog [4]

in Sect. 3 the translation of the Jason constructs, the Jason reasoning cycle, the process orchestration of eJason, and the current limitations of the approach are explained. Some early benchmarks results for the eJason prototype are reported in Sect. 4. Finally, a summary of our conclusions and items for future work appear in Sect. 5.

## 2 Erlang

Erlang [3, 10] is a functional concurrent programming language created by Ericsson in the 1980s. 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. In this section we briefly describe the key aspects of Erlang.

### 2.1 Functional Erlang

In Erlang basic values are: integers, floats, atoms (starting with a lowercase letter), bit strings, binaries, and `funs` (to create anonymous functions), and process identifiers (pids). The compound values are lists and tuples. Erlang syntax includes a record construct which provides syntactic sugar for accessing the elements of a tuple by name, instead of by position. Functions are first class citizens in Erlang. For example, consider the declaration of the function `factorial` that calculates the factorial of a number.

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

As in Prolog, variable identifiers (`N`) start with a capital letter, and atoms (`factorial`) with a lowercase letter. Like Prolog, Erlang permits only single assignment to variables.

As virtually all functional programming languages, Erlang supports higher order functions.

### 2.2 Concurrent and Distributed Erlang

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, with a unique node name. Communication is asynchronous and point-to-point, with one process sending a message to a second process identified by its pid. Messages sent to a process are put in its message queue, also referred to as a mailbox. Informally, a mailbox is a sequence of values ordered by their arrival time. Mailboxes can in theory store any number of messages. Although mailboxes are ordered, language constructs permit retrieving messages from the process mailbox in arbitrary order.

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,

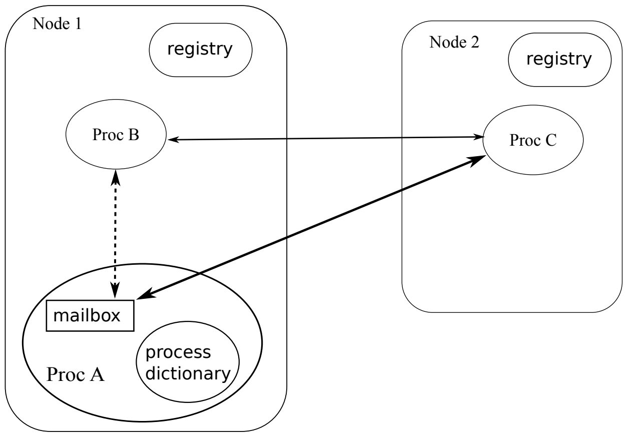

**Fig. 1.** An Erlang multi-node system

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 pid registry is local to a node.

A unique feature of Erlang that greatly facilitates building fault-tolerant systems is that processes can be "linked together" in order to detect and recover from abnormal process termination. If a process $P_1$ is linked to another process $P_2$, and $P_2$ terminates with a fault, process $P_1$ is automatically informed of the failure of $P_2$. It is possible to create links to processes at remote nodes.

As an integral part of Erlang, the OTP library provides a number of very frequently used design patterns (behaviours in Erlang terminology) for implementing robust distributed and concurrent systems. The most important OTP design patterns are generic servers that implement client/server architectures, and supervisors, to build robust systems. Other OTP design patterns implement, for instance, a publish-subscribe mechanism, and a finite state machine.

## 3 Implementing Jason in Erlang

This section describes the implementation of a subset of Jason in Erlang.

### 3.1 A simple running example in Jason

To illustrate the implementation of Jason in Erlang, we use the example in Fig. 2, which illustrates the main syntactical elements of the Jason language.

This somewhat artificially programmed agent is a *counter*, which prints a message when finished. The initial beliefs of the agent are (a), representing that

```
init_count(0).                                              (a)
max_count(2000).                                            (b)

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

!startcount.                                                (d)

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

+!count: actual_count(X) & max_count(Y) & X < Y <-          (f)
                         ?next(X, NewCount);
                         -+actual_count(NewCount);
                         !count.


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

**Fig. 2.** A simple Jason agent

the agent believes the initial value to be zero, and (b), representing that the agent believes that it has to count up to 2000. There is one rule (c), expressing the successor relation for numbers. The agent's initial goal (d) is to start counting. There are three plans (e), (f) and (g). Plan (e) initializes the actual counter by adding a new belief to the agent's belief base and introduces a new achievement goal `!count`. That goal can be achieved by plans (f) and (g), whose context's are disjoint and, thus, can never be considered as applicable plans at the same time. When plan (g) is executed, which occurs when the agent has counted up to its limit, it prints a message and the agent remains waiting as there are no more events. We kindly direct the reader to [9] for a complete definition of the Jason programming language and its interpreter, as a detailed description of the different features of Jason lies beyond the scope of this paper.

### 3.2 An overview of the implementation

Jason is both a programming language which is an extension of AgentSpeak, and an interpreter of this programing language in Java. The constructs of the Jason programming language can be separated into three main categories: beliefs, goals and plans. The Jason interpreter runs an agent program by means of a reasoning cycle that provides the operational semantics of the agent. This semantics has been formalised and can be found in [9].

The translation of beliefs and goals to Erlang is rather straightforward since they represent the knowledge of an agent, rather than its behaviour (with the exception of rules). Common Erlang data types and functions are used to translate these Jason constructs to Erlang. Initially we used the third party software

ERESYE [15] (ERlang Expert SYstem Engine) to implement the belief base of each agent. 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. We decided to use this software as the term storage service due to its capabilities to store Erlang terms and to also retrieve them using pattern matching. Nevertheless, due to the way in which we used this software, the resulting Jason implementation was rather inefficient. Therefore we decided to implement our own belief base. This later implementation represents the belief base of each agent as a list of ground terms. The translation of beliefs, goals and rules to Erlang is explained in Sect. 3.3

The implementation of plans is more convoluted due to their dynamic nature. Every plan is composed by one or more formulas that must be evaluated sequentially. However, the formulas in a plan may not all be executable in the same reasoning cycle. The representation of plans in Erlang, and their execution by a tail-recursive Erlang function, is explained in Sect. 3.4.

A higher-level view of the different Erlang *processes* implementing the Jason reasoning cycle [9] and the communication between them is described in Sect. 3.5, while Sect. 3.6 provides the details. Basically, the reasoning cycle of each agent is handled by a different Erlang process.

Finally Sect. 3.7 enumerates the limitations of eJason, with respect to implementing the full Jason language, at the time of writing this paper.

### 3.3 Translation of Jason beliefs and goals into Erlang

Here we describe how the different constructs for representing and inferring knowledge of Jason are implemented.

**Variables.** To represent the bound and unbound variables of a plan we use a variable valuation that is updated as variables become bound to values. Concretely, a valuation for a plan is represented by an Erlang tuple where values are associated with distinct variables ordered according to the order in which these variables first occur in the plan. For instance, a possible valuation for the second plan `(f)` in Fig. 2 would be $\{0, 2000,$ `'_'`$\}$, thus binding `X` to 0, `Y` to 2000 and leaving `NewCount` unbound.

**Beliefs.** Every agent possesses its own belief base, i.e., each agent can only access and update its own belief base. In a first version of eJason, we used ERESYE in the following manner. Each agent ran its own ERESYE engine, which spawned three Erlang processes for each belief base. Early experiments showed that this implementation was rather inefficient. For instance, the eJason implementation of the counter example could only handle around four thousand agents. An alternative is to use a single ERESYE engine for all agents, and provide some means to isolate the beliefs of each agent from everyone else's. We discarded this approach because the autonomy of agents would have been compromised.

For instance, a failure in the ERESYE engine would cause a failure in the belief base of all agents. Therefore, we decided to implement our own belief base in a separate module, named *beliefbase*, which provides the functionality to access and update a belief base without having to create a separate Erlang process. As explained earlier, this belief base is represented as a list of Erlang terms, where each term in the list corresponds to a different belief.

A belief, i.e., either an atom or a ground formula, is represented in eJason as an Erlang tuple. An atom belief is represented in Erlang as the tuple containing the atom belief itself, e.g., {*atom_belief*}. A ground formula belief is represented by an Erlang tuple with three elements. The first element is the name of the predicate, the second is a tuple which enumerates the arguments of the predicate, and the third is a list containing a set of annotations. Each annotation can either be an atom or a predicate and is represented in the same manner as a belief. As an example, the belief base of the running example (with an added annotation):

```
init_count(0).
max_count(2000)[source(self)].
```

is translated to the following Erlang term:

```
{ init_count, {0}, [] }.
{ max_count, {2000}, [ {source,{self},[]} ] }.
```

**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.

**Goals.** Goals are represented in the same way as beliefs. Nevertheless, they are never stored in isolation, but as part of the body of an event, as specified below.

**Events.** As we have not yet implemented perception of the agent environment, events always correspond to the explicit addition or deletion of beliefs, or the inclusion of achievement and test goals. An event is composed of an event body, an event type, and a related intention. The event body is a tuple that contains two elements. The first element is one of the atoms {*added_belief, removed_belief, added_achievement_goal, added_test_goal*}. The second element is a tuple that represents the goal or belief whose addition or deletion generated the event. The event type is either the atom *internal* or the atom *external*, with the obvious meaning. The related intention is either a tuple, as described below, or the atom *undefined* to state that the event has no related intention. The only internal events that possess a related intention are the events corresponding to the addition of goals, as their intended means will be put on top of that intention. The intended means for the rest of events will often generate new intentions. When a relevant plan for the event is selected, the list of Erlang functions that execute the formulas in its body is added either on top of a related intention or as a brand new intention (e.g. in the case of external events). For instance, consider the following formulas in the body of a plan belonging to some intention *Intention*:

```
+actual_count(NewCount);
?next(X, NewCount);
```

The events generated after their respective execution would be:

```
{event, internal, {added_belief,
                   {actual_count,{NewCount}, []} }, undefined }

{event, internal, {added_test_goal,
                   {next, {X, NewCount}, []} }, Intention}
```

For the sake of clarity, the variable *Intention* appears as placeholder for the real representation of the corresponding related intention.

## 3.4 Implementing Jason plans in Erlang

**Body of a plan.** Every Jason plan is composed by one or more formulas that must be evaluated in a sequence. However, these formulas are not all necessarily evaluated during the same reasoning cycle of the agent, e.g., due to the presence of a subgoal that must be resolved by another plan. To be able to execute the formulas separately, each formula is implemented by a different Erlang function. Then, the representation of the body of a Jason plan is a list of Erlang functions. Each of these functions implements the behaviour of a different formula from the Jason plan. The order of these functions in the list is the same order of the body formulas they represent. The implementation and processing of the formulas in a plan body is the most intricate task in the implementation of Jason in Erlang.

**Plans.** A Jason plan is represented by a record having three components: a *trigger*, a *context* and *body*. The trigger element is a function which, applied to the body of an event, returns either the atom *false* if the plan does not belong to the set of relevant plans for the event or the tuple {*true,InitialValuation*}, where *InitialValuation* provides the bindings for the variables in the trigger. The context is a function which, when applied to the initial valuation obtained from the trigger, returns a list of all the possible valuations for the variables in the trigger and context that satisfy the context. Finally, the body element is the list of Erlang functions that implement the body of the plan, as described before.

As an example, consider the plan for agent *counter*:

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

The *plan* record generated for the plan above is

```
{plan, fun start_count_trigger/1,
       fun start_count_context/1,
       [Fun1, Fun2]}
```

where the `start_count_trigger/1` and `start_count_context/1` functions implement the trigger and the context respectively. The list at `[Fun1,Fun2]` represents the plan body, where `Fun1` implements the formula `+actual_count(X)` and `Fun2` implements the formula `!count`.

**Intentions.** The stack of partially instantiated plans that compose each of the Jason intentions is represented as a list of Erlang records. Each of these records is composed of four elements. The first element is the event that triggered the plan. This element is kept as a meta-level information that can be accessed by the intention selection function. For instance, we could give priority to the execution of intentions whose partially instantiated plan on top of the stack resolves a test goal. The second element is the plan record chosen by the option selection function and, again, is intended to serve as a meta-level information accessible by the intention selection function. The third element is a tuple that represents the intended means of the intention plan, i.e. the bindings for the variables in the partially instantiated plan. The fourth element is a list of Erlang functions, representing the formulas of the partially instantiated plan that have not been executed yet. If an intention is selected for execution, the record for the partially instantiated plan on top of it (i.e. the first element of the list that represents the intention) is obtained. Then, the function at the head of the list of Erlang functions in the fourth element of this record is applied to the current variable valuation. Finally, this last function is removed from the list. This process amounts to processing the formula on top of the intention stack as is required by the specification in [9].

**Selection functions.** The event, plan and intention selection functions for a MAS can be customised by providing new implementations (in Erlang) of the functions `selectEvent`, `selectPlan` and `selectIntention`.

## 3.5 Process Orchestration and Communication

The multiagent system generated by the translation from Jason to Erlang maps each agent to an Erlang process, all executing on the same Erlang node. Each Erlang agent process can be accessed using either its process identifier, or the name of the Jason agent. The name of the agent is associated with the Erlang process using the Erlang process registry. In case multiple agents are created with the same name an integer (corresponding to the creation order) is appended to the registered name to keep such names unique. An item for future work is to extend Jason with new mechanisms to create multiple agents from the same agent definition, and to associate symbolic names with such agents, as the present mechanisms are somewhat unwieldy.

The communication between agents is implemented using Erlang message passing. As an example, consider a system where the agent *alice* sends different messages to agent *bob* by executing the internal action formulas:

```
.send(bob, tell, counter(3)}

.send(bob, untell, price(coffee,300))

.send(bob, achieve, move_to(green_cell))
```

The actions are mapped to the following Erlang expressions:

```
bob ! {communication,alice,{tell,{counter,{3},[]}}}.

bob ! {communication,alice,{untell,{price,{coffee,300},[]}}}.

bob ! {communication,alice,{achieve,{move_to,{green_cell},[]}}}).
```

The Erlang expression `Receiver ! Message` deposits `Message` into the mailbox of agent `Receiver`. The atom *communication* is used to declare the message type. It is included in the implementation to enable processes to exchange other types of messages, possibly not related to agent communication, in a future extension of eJason.

Agent *bob* can process the different messages sent by *alice* by checking its mailbox, which is performed automatically in every iteration of the reasoning cycle. The Erlang expression that retrieves the message from the process mailbox:

```
receive
  {communication,Sender,{Ilf,Message}} ->
    case Ilf of
      tell ->     ... %% Process tell message
      untell ->   ... %% Process untell message
      achieve ->  ... %% Process achieve message
    end.
```

These examples show how easily the agent communication between Jason agents can be implemented in Erlang. The simple yet efficient process communication mechanism of Erlang is one of the principal motivations to implement Jason agents using the Erlang programming language.

In the example above, all the agents are located in the same MAS architecture; messaging between agents in different architectures would be easy to support too, and would not require the use of a communication middleware like SACI. However, such an extension is not yet implemented in eJason.

## 3.6 Representing the Jason Reasoning Cycle in Erlang

The Jason reasoning cycle [8] 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:

```
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.
```

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 Jason 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.

## 3.7 Jason Subset Currently Supported

eJason currently supports only a subset of the Jason constructs needed to implement complex multi-agent systems. However, we foresee no major difficulties in adding the additional features not currently supported, and expect to do so in the near future. The features of the Jason language not currently supported are the following:

1. **Belief annotations.** Even though our Jason parser accepts code with belief annotations, these annotations are not taken into account when resolving plans (e.g., when checking whether a plan context is satisfied).
2. **Annotations on plan labels.** The meta-level information associated with plans is removed during the lexical analysis.
3. **Plan failure handling.** Whenever a plan fails, e.g., because test goal in the plan body cannot be successfully resolved, the whole intention that the plan belongs to is dropped. Moreover, no new event is generated as a result of the plan failure.
4. **Environment.** The environment of eJason programs is not currently modelled. Therefore, no external actions, except console output, are allowed and no perception phase is required.
5. **Distribution.** There is no support for distributed agents.
6. **Communication.** The only illocutionary forces that are properly processed are *tell*, *untell* and *achieve*. Messages with any other kind of illocutionary force are ignored and dropped from the mailbox of the agent.
7. **Library of internal actions.** The only internal actions considered are ".print" (to display text on the standard output) and ".send" (to interact with other agents in the same multiagent system).
8. **Unbound plan triggers.** The trigger of every plan must be either an atom or a predicate (whose parameters do not need to be bound) but never a variable.
9. **Decomposition operator.** The binary operator "=..", used to (de)construct literals (i.e. predicates and terms), is not accepted by the parser.
10. **Code order.** The grammar accepted by the parser is similar to the simplified one presented in Appendix 1 of [9]. Therefore, the source code to be translated must state first the initial beliefs and rules, followed by the initial goals and, finally, the different plans.
11. **Multiagent system architecture.** There is only one kind of agent infrastructure implemented. It runs all the agents in a multiagent system within the same Erlang node.

## 4 Experiments

To test the performance of eJason, we use two simple Jason programs. The first is the counter example of Fig. 2 in Section 3.1. The second represents an agent that outputs two greeting messages on the console. To add some complexity to the behaviour, the contents of those messages are obtained from the set of beliefs of the agent using queries which have both bound and unbound variables. The examples were run using different numbers of homogeneous agents, i.e., all the agents behaved the same. All of them were run under Ubuntu Linux version 10.04 in a computer with two 2.53 GHz processors. With these examples, we want to measure the execution time of the generated MAS and their scalability with respect to the number of agents in the system.

The preliminary results are presented in Tables 2 and 3.

| Number of Agents | Jason Execute Time (magnitude) | eJason Execution Time (milliseconds) |
|---|---|---|
| 10 | seconds | 20 |
| 100 | minutes | 500 |
| 1000 | not measurable | 1181 |
| 10000 | not measurable | 7916 |
| 100000 | not measurable | 18674 |

**Table 1.** Execution times for the pupil-teacher multiagent system

***BORRAR
**** END BORRAR

| Number of Agents | Jason Execute 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

The results indicate that the multiagent systems generated by eJason scale to some hundreds of thousands of agents with an average execution time of a few seconds. Regarding the multiagents systems generated by Java-based Jason, we can see that they required more time to execute (the exact time quantities could not be precisely measured) and that it was not possible to increase the number of agents over a few thousands (in the cases labeled as *not measurable* a java.lang.OutOfMemoryError exception was raised).

Clearly these are only preliminary findings as more thorough benchmarking is needed.

## 5 Conclusions and Future Work

In this paper we have described a prototype implementation of eJason, an implementation of Jason, an agent-oriented programming language, in the Erlang concurrent functional programming language. The implementation was rather straightforward due to the similarities of Jason and Erlang. eJason is able to generate Erlang code for a significant subset of Jason. Early results are promising, as the multiagent systems running under the Erlang runtime system can

| Number of Agents | Jason Execute 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

make use of the Erlang lightweight processes to compose systems of thousands of agents, where the process generation, scheduling, and communication introduce a negligible overhead. We also describe and motivate some of the implementation decisions taken during the design and implementation phases, such as e.g. the use of the ERESYE tool during an early stage and its later replacement.

Clearly, the similarities between the capabilities of agents and the Erlang processes are many, with the exception of the support for programming rational reasoning in Jason. We believe that the existence of eJason can help attract Erlang programmers to the MAS community, by providing them a convenient and largely familiar platform in which to program rational agents, while being able to implement the rest (adapting interpreter meta behaviour, and actuators for the environment) in Erlang itself. Moreover we believe that the MAS community can benefit from having access to the efficient concurrency and distribution capabilities of Erlang, while maintaining backward compatibility with legacy code, and without the need to develop a new agent-based language.

Clearly, as eJason is still a prototype, there are numerous areas for future work and improvement. The subset of Jason implemented at the moment is quite small; it is, for example, necessary to add support for belief annotations and plan labeling. Moreover, we plan to add support in eJason for different distributed agent architectures. An essential item for near future work is the implementation of a means for agents to act on their environment. We intend to make eJason agents capable to cause changes in their environment using actions programmed either in Java or Erlang, i.e., there should be no need to rewrite the large body of existing Java code for Jason environment handling. Besides, we expect to be able to use the agent inspection mechanisms already implemented in e.g. JEdit.

Another item for future work includes prototyping extensions to Jason; we believe that eJason is a good platform on which to perform such experiments. Finally we also intend to experiment with model checking, applied on the resulting Erlang code, to verify Jason multiagent systems.

## References

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

2. *Simple Agent Communication Infrastructure. See: http://www.lti.pcs.usp.br/saci/.*
3. Joe Armstrong. Programming Erlang: Software for a concurrent world). The Pragmatic Bookshelf, 2007.
4. Joe Armstrong, Robert Virding, and Mike Williams. Use of Prolog for developing a new programming language. In *Proc. of the international conference on Practical Application of Prolog*, 1992.
5. Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. JADE - a Java agent development framework. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, 2005.
6. Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Wiley, April 2007.
7. R. Bordini and J. Hübner. Bdi agent programming in agentspeak using jason. *Computational logic in multi-agent systems*, pages 143–164, 2006.
8. Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12:2006, 2006.
9. Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
10. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.
11. J.F. Hübner. *Um modelo de reorganização de sistemas multiagentes [In Portuguese]*. PhD thesis, Universidade de São Paulo, Escola Politécnica, Brazil, 2003.
12. P. D. O'Brien and R. C. Nicol. FIPA - Towards a Standard for Software Agents. *BT Technology Journal*, 16:51–59, July 1998.
13. Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 January 1996, Proceedings.
14. Anand S. Rao and Michael 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.
15. Antonella Di Stefano, Francesca Gangemi, and Corrado Santoro. ERESYE: artificial intelligence in erlang programs. In Konstantinos F. Sagonas and Joe Armstrong, editors, *Erlang Workshop*, pages 62–71. ACM, 2005.
16. M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.
17. Michael Wooldridge. Reasoning about rational agents. MIT Press, 2000.