Notes on pragmatic agent-programming with Jason

Radek Píbil[1,2], Peter Novák[1], Cyril Brom[2], and Jakub Gemrot[2]

[1] Agent Technology Center, Department of Computer Science and Engineering
Faculty of Electrical Engineering, Czech Technical University in Prague
Czech Republic
[2] Department of Software and Computer Science Education
Faculty of Mathematics and Physics, Charles University in Prague
Czech Republic

Abstract *AgentSpeak(L)*, together with its implementation *Jason*, is one of the most influential agent-oriented programming languages. Besides having a strong conceptual influence on the niche of BDI-inspired agent programming systems, *Jason* also serves as one of the primary tools for education of and experimentation with agent-oriented programming. Despite its popularity in the community, relatively little is reported on its practical applications and pragmatic experiences with adoption of the language for non-trivial applications.
In this paper, we present our experiences gathered during an experiment aimed at development of a non-trivial case-study agent application by a novice *Jason* programmer. In our experiment, we tried to use the programming language *as is*, with as few customisations of the *Jason* interpreter as possible. Besides providing a structured feedback on the most problematic issues faced while learning to program in *Jason*, we informally propose a set of ideas for solving the encountered design problems and programming language issues.

1 Introduction

Jason [8] is an agent-oriented programming system implementing the agent programming language *AgentSpeak(L)* [19]. *AgentSpeak(L)* was proposed as a theoretical language, an articulation and operationalization of the Bratman's Belief-Desire-Intention architecture [9]. *Jason* is nowadays one of the popular approaches in the group of theoretically-rooted agent-oriented programming languages (APLs). Some other members of this group include also *2APL*, *3APL*, *GOAL*, *Golog*, *Jazzyk*, etc. (for an overview consult e.g., [5,7,6,16]). Building on the foundations of formal logics, these languages serve as vehicles for study of both theoretical issues in agent systems (language features, generic programming constructs, reasoning, coordination, etc.), as well as practical aspects of their design and implementation (e.g., modularity, design, debugging, or code maintenance). To enable program verification, or model checking for more rigorous reasoning about agent programs, *Jason*, together with the majority of APLs

in this class, puts a strong emphasis on their rooting in computational logic and rigorous formal semantics. Unlike the more pragmatic approaches, such as *Jadex*, or *JACK* (cf. [18,20]), these APLs were designed from scratch. While providing the advantages we have discussed, this has also created serious shortcomings with respect to the practicality of their use, such as those discussed in this paper.

On one hand, pragmatic problems of agent design and implementation, such as code modularity, are gaining a more prominent role in the research community. On the other hand, a feedback on practical use of such APLs in more elaborated settings is rather scarce. *AgentSpeak(L)* often serves as a basic APL for various extensions and integration with 3rd party tools. However, little is reported on its practical applications and experiences with its use, be it in more involved applied research projects, or in more significant close-to-real-world applications (cf. also the *Jason* related projects website [14]). To date, the only report on pragmatic issues of *Jason* in a more involved context is the recent study by Madden and Logan [15] in which the authors deal with problems of modularity in their application and in turn propose corresponding improvements of the language itself. At the same time, to our knowledge, the most elaborated applications of the *Jason* programming system include the entries to the *Multi-Agent Programming Contest*, which already witnessed eight submissions in years 2006-2010 altogether by three independent research groups [2,1]. The reports on development of these applications do not include a discussion of practical issues of an agent program implementation, but rather focus on the analysis and design aspects with an emphasis on the multi-agent coordination.

In this paper we discuss our experiences gathered during an experiment aimed at developing a non-trivial case-study multi-agent application by a novice *Jason* programmer. The main goal of the undertaking was an exploration of basic problems in multi-agent coordination in a simple simulated environment using the *Jason* programming system. In particular, we have implemented an application involving a team of eight agents collaboratively exploring a grid maze and subsequently traversing the environment while cooperatively maintaining a formation. Our experiment aimed at a naïve, and relatively conservative use of the *Jason* programming system. We tried to use the programming language *as is*, with as few customisations of the *Jason* interpreter as possible. In contrast, most involved example applications published at the *Jason* project website [14] and submissions to the *AgentContest* employ extensive customisations of the *Jason* interpreter as an inherent part of the system implementation.

The contribution of the presented paper is twofold. Firstly, we provide a structured feedback on the most problematic issues faced while learning to program in *Jason*. Secondly, without an ambition to provide conclusive technical solutions, we rather informally propose a set of ideas aimed at solving the discussed design problems and programming language issues.

After a brief introduction of *AgentSpeak(L)* and *Jason* in Section 2 the subsequent Section 3 provides a description of the implemented case-study. In Section 4, the core of this paper, we discuss a selection of problems we have faced

during the experiment. For each discussed issue, we firstly motivate and explain the problem on the background of the introduced case-study application, or its extension, and then we discuss possible solutions. The topics covered in the discussion include implementation of a simple loop design pattern, handling interactions between several plans and interruptibility thereof, and usage of mental notes as local variables in plans. We also discuss two technical issues arising from implementation of agents embodied in dynamic environments and the unclear boundary between *Jason* programming language itself and its underlying customisation API in *Java*. We conclude the paper by final remarks in Section 5.

2 AgentSpeak(L) and Jason

AgentSpeak(L) is a theoretical agent-oriented programming language introduced by Rao in [19]. It can be seen as a flavour of logic programming implementing the core concepts of the BDI agent architecture, a currently dominant approach to design of intelligent agents. Structurally, an *AgentSpeak(L)* agent is composed of a *belief base* and a *plan library*. The belief base, essentially a set of belief literals, provides the initial beliefs of the agent. The plan library serves as a basis for action selection, as well as for steering the evolution of the agent's mental state over time. The plans of the agent are rules of the form `event : context` $\leftarrow$ `plan`. The rule denotes a plan, a sequence of basic actions and/or subgoals, which is applicable in reaction to the triggering event if the context condition, a conjunction of belief literals, is satisfied.

AgentSpeak(L) agents are reactive planning systems which react to events occurring in their environment, or are generated as subgoals internally by the agent as a result of a deliberative change in its own goals. The dynamics of the agent system is facilitated by i) instantiation of abstract plans as intentions relevant in particular contexts, and subsequently ii) gradual execution of the intentions leading to their subsequent decomposition into more and more concrete subgoal invocations and finally atomic action executions. In each deliberation cycle, such an agent performs the following sequence of steps:

1. *perceive* the environment and update the belief base accordingly,
2. *select an event* to handle,
3. retrieve all *relevant plans*,
4. *select an applicable plan* and *update the intentions* accordingly,
5. *select an intention* for further execution,
6. *execute one step* of an intention and modify the intention base and the set of events accordingly.

Jason is a *Java*-based programming system implementing *AgentSpeak(L)* with various extensions. It also includes an integration with several multi-agent middleware platforms such as *JADE*, or *Moise+*. In its original incarnation, *AgentSpeak(L)* is underspecified in several points of the deliberation cycle. In particular, in how exactly the three selection functions $\mathcal{S}_\mathcal{E}$, $\mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{I}$, denoting

the selection of events, applicable plans and intentions respectively, are implemented. In *Jason*, these are customizable functions that can be implemented as *Java* methods. Furthermore, *AgentSpeak(L)* disregards the implementation details of agent's interaction with its environment. That is, the interpreter assumes that the belief base was updated according to agent's percepts at the beginning of each deliberation cycle. *Jason* extends the framework for reasoning about agent's beliefs in that it incorporates a *Prolog* interpreter for the belief base and also provides a toolbox for implementation of custom belief bases, such as the topology of environments, or interface to relational databases. Finally, *Jason* provides a framework for an implementation of perception handlers and external events as *Java* methods, together with an API for implementation of customised exogenous actions embodying the behaviours of the agent in its implementation.

The customisation interfaces of the *Jason* interpreter provide means to tailor the deliberation cycle to the domain specific requirements, as well as to improve the efficiency of the agent program execution. Our motivation in the presented experiment was to explore the issues faced in the course of agent program implementation using the vanilla *Jason* interpreter. The main requirement underlying the experiment was to make only minimal customisations of the interpreter required to make the implemented agents interact with their environment.

3 The case-study

The *Cows & Cowboys* problem of the *Multi-Agent Programming Contest* editions 2009 and 2010 (cf. [4], scenarios for the 2009-10 editions) is a challenging scenario for cooperative multi-agent teams benchmarking. In the *Cows & Cowboys* scenario, two teams of agents, herders, compete for a shared resource, cows. The environment is a grid, usually a square with a side approximately 100 cells. Each cell can be either empty, or can contain an object which can be either a tree, a fence, an agent, or a cow. Trees serve as obstacles in the environment and are arranged so that the freely traversable space forms a kind of a maze. The agents can move between empty cells and can open fences, by standing at the edge of a fence. Similarly to the agents, cows also roam around through empty cells, however their movement is controlled by the environment. It takes into account their mutual distances, as well as distances from the agents and trees the cow can see. The agents and cows have a limited view, and in each simulation step receive a perception containing cells in their vicinity. The task of each agent team is to herd as many cows as possible into a corral belonging to the team. Because cows are afraid of the agents, they can be pushed by a coordinated movement of a team of agents.

For purposes of this case-study, we have implemented a fragment of the *Cows & Cowboys* scenario. The concrete problem was to implement a team of agents, which cooperatively explore the maze, find some pre-determined landmarks and then traverse the maze from one landmark to another while maintaining a formation of a particular shape.

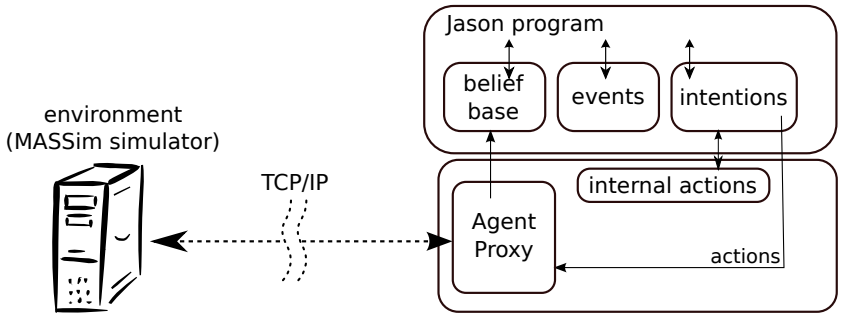

Figure 1. The architecture of a single *Jason* agent interacting with the simulated environment.

The simulated environment was provided by the MASSim server [3]. The architecture of the implemented system is depicted in Figure 1.

During every simulation step the belief base is updated and an action from the previous timestep is marked as executed, if there was any. *Jason* thread is then allowed to continue its deliberation based on new percepts. Subsequently, the agent's control thread goes to sleep for 2000 milliseconds (the server sends new percepts every 2500 milliseconds) unless it is woken up by the *Jason* thread upon an invocation of an exogenous action from within an intention of the agent. Finally, the indicated action to perform is validated by checking whether it is intended for the current timestep and if found valid, it is sent back to the server. The only exogenous actions the agent can execute are moves in the eight directions: *north*, *east*, *south*, *west* and the diagonal moves *north-east*, *north-west*, *south-east* and *south-west*.

The toolbox of internal actions includes most importantly the implementation of the path planning algorithm A*, together with a few auxiliary functions such as a lottery-like mechanism for choosing the formation leader, queries for contents of map cells, etc.

One of the most important decisions for the implementation of the case-study was that we did not customise the *Jason* interpreter itself, nor the event plan intention selection functions $\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{P}}$, $\mathcal{S}_{\mathcal{I}}$.

4 Issues faced

In the following, we discuss a set of problems we encountered in the course of implementing the case-study described above in Section 3. The programmer involved in the experiment was new to BDI-style agent-oriented programming and was learning the *Jason* language along the way. We used the book *Programming Multi-Agent Systems in AgentSpeak Using Jason* [8] as the authoritative source and documentation for *Jason*. For clarity, the discussion of each issue includes a brief motivation and explanation of the particular design problem, sub-

sequently followed by a discussion on the available solutions, their consequences and wherever appropriate an informal proposal for an improved solution to the issue.

4.1 Loop implementation

Quite often a programmer needs to implement some kind of a loop design pattern. In a maze-like environment, the agent calculates a path from point A to point B using a path planning algorithm and then it follows the path. This pattern could be implemented by the following algorithm in an imperative language:

```
before−loop−code
while not loop−condition do
        loop−body
end
after−loop−code
```

As of conducting the here reported experiment, *Jason* did not feature a loop programming construct per se, but it could be implemented by the following *Jason* code:

```
event: context ←
        !before−loop−plan;
        !loop;
        !after−loop−plan.
+!loop: not loop−condition ←
        !loop−body;
        !loop.
```

This pattern implements the idea of tail recursion. The interpreter does not feature a special treatment of tail recursion though. According to the language semantics, this pattern unfolds into a growing intention stack. At the bottom of the stack there is the `after−loop−plan` goal. Above it in the stack, there is a series of invocations of `!loop` of a length equal to the number of iterations of the loop. In order to facilitate correct plan failure handling, *Jason* interpreter does not remove the top-level invocation from the intention stack. In the path-following scenario, if the path is of length 1000, the intention stack would grow to the size 1000 plus the length of `after−loop−plan`. Notice that several dozen thousands path steps are not that unrealistic for large grid environments. In cases with an extremely high number of loop iterations, the intention stack growth can lead to a high memory consumption. Perhaps even more importantly, the following clean-up of the intention stack, may take an undesirably long time. The execution of `after−loop−plan` may therefore be heavily delayed. Possibly even missing some important timing window. This issue is the same as with the depth-first search algorithm (DFS). There are two main approaches to the DFS implementation: an exclusive stack and a recursive function call. The exclusive stack solution requires only the to-be-explored nodes, while the recursive function call solution requires the activation records of the recursive function to be present on the program stack as well.

A naïve attempt by a novice programmer could be a loop implementation using the asynchronous goal invocation `!!loop`. A straightforward application is

inappropriate in this context though as besides invoking the loop, it would lead to an immediate continuation with the after−loop−plan.

We propose the following implementation of the loop design pattern, which uses higher order variables feature of *Jason* (cf. [8], Chapter 3) to implement a kind of a callback scheme:

```
event: context ←
        before−loop−plan;
        !!loop(after−loop−event).
+!after−loop−event: true ← after−loop−plan.
+!loop(Callback): not loop−condition ←
        loop−body;
        !!loop(Callback).
+!loop(Callback): loop−condition ← !!Callback.
```

The above loop implementation is well-formed and a valid program according to the *Jason* syntax and semantics. Instead of a synchronous event invocation, we invoke the loop in an asynchronous manner !!loop and provide it with an argument, which is a string denoting the event, which should be invoked after the loop finishes – in this case after−loop−event. When the loop termination condition becomes true, the pattern simply invokes the event stored as the callback. The advantage of this loop implementation is that it does not lead to an intention stack growth, while at the same time still allows for plan failure handling as in the standard loop implementation.

In the pattern above, the loop has a callback argument. This callback is added as a goal upon the loop's successful termination. An extension of this callback design solution allows a programmer to introduce a powerful plan failure handling mechanism as follows:

```
event: context ←
        before−loop−plan;
        !!loop(after−loop−event, fail−loop−event).
+!after−loop−event: true ←
        after−loop−plan.
+!fail−loop−event: true ←
        loop−failure−plan.
+!loop(SuccessCallback, FailCallback): not loop−condition & loop−continuation−condition ←
        loop−body;
        !!loop(SuccessCallback, FailCallback).
+!loop(_, FailCallback): not loop−condition & not loop−continuation−condition ←
        !!FailCallback.
+!loop(SuccessCallback, _): loop−condition ←
        !!SuccessCallback.
```

A loop is a handy and a frequently used design pattern in imperative programming. However, for a novice programmer, a loop implementation in *Jason* is rather unintuitive and it often leads to a confusion. One of the straightforward solutions, well in the spirit of BDI architecture, would be to use persistent goals, such as in 3APL. Another way to deal with this would be to implement a built-in loop programming construct, or a macro pre-processor construction similar to the various types of goals and commitment strategies discussed in [8], Chapter 8, or in [17].

To conclude, in the course of writing up and submission process of this paper, a new version of *Jason* interpreter was released. In the most recent version of the interpreter (from ver. 1.3.4 on), *Jason* includes a loop construct in the form

of two internal actions for (foreach) and while. As a result, this point is no longer a pressing issue for *Jason*, yet the more general solution of the problem presented above might come handy as a standalone pattern.

4.2 Interruptions and intention interactions

Among other desirable properties, intelligent agents are supposed to be able to follow long term goals, but at the same time should be reactive to events in the environment and proactively seek opportunities for action whenever they arise in an appropriate context. Consider the following slight extension of the case-study scenario. The team of agents is moving through the environment in a formation, however, agents are also capable of picking up objects, let's say garbage, from the cells they stand on. Let's also assume, an agent perceives the object to pick, only when it is located in the same cell as the object and it can pick up an object only after it closely inspected it. In *Jason*, a straightforward and naive implementation of the two behaviours would look like as follows:

```
+!formation_loop : not aligned ←
        /* calculate the move action towards formation position */
        move;
        !formation_loop.
+see(Object) : true ←
        inspect(Object);
        pick(Object).
```

The above naive implementation does not work properly using the vanilla *Jason* interpreter. The reason is that after the new intention leading to picking up the object from the cell is formed, it is not ensured that in the same deliberation cycle, the intention selection function $\mathcal{S}_{\mathcal{I}}$ selects the same intention for execution. In the case $\mathcal{S}_{\mathcal{I}}$ selects for execution first the intention for keeping the formation aligned, it can happen that at the moment the agent wants to inspect, or pick up the object, the plan fails since the agent is no more located in the same cell as the object – the plan for keeping the formation aligned moved it away.

The implementation problem described above is that of interacting intentions (run-time plans) that can mutually interrupt each other. In *Jason*, similarly to most state-of-the-art BDI-based agent programming languages, intentions are implicitly considered interruptible. However, having several intentions involved in the same context, i.e., modifying the same aspect of agent's state, which can be instantiated as intentions in parallel, the problem is *how to determine the priority of execution of the corresponding intentions?* Below, we discuss several different solutions to this problem.

A straightforward approach would be to use some kind of plan synchronisation mechanism. *Jason* provides atomic, a pre-defined plan annotation construct ensuring that the intention instantiated from an atomic plan is executed without interruption until it finishes. The following code presents the use of this construct:

```
@object_picking[atomic]
+see(Object) : true ←
        inspect(Object);
        pick(Object).
```

While simple and straightforward, this solution of the plan interaction does not scale with the number of involved interacting intentions. Consider that our agent should be able to quickly renegotiate the details of formation location and its heading with the team. While interdependent with the formation alignment behaviour, it is independent to the object picking behaviour. In result, we would like to impose the following ordering on the three behaviours: the formation alignment behaviour is preceded by the opportunistic object picking, which is in turn preceded by the negotiation. However, the atomic construct applied to the object picking behaviour would cause it to be non-interruptible, hence the negotiation could not take place.

Another possibility to deal with interacting intentions would be to let the program handle the situations, in which they can be interrupted, not the intentions themselves. I.e., all plans would be considered implicitly non-interruptible and at every point in which an intention can be interrupted by a higher-priority event, there would be an explicit check for all possibilities of such interruptions, followed by a synchronous invocation of the interrupting event and an explicit check for preconditions of the remaining plan. The following code snippet demonstrates a usage of such a technique:

```
+!formation_alignment : context ←
        align−plan−start;
        !pick_object; !negotiation;
        align−plan−rest.
+!pick_object : see(Object) ←
        pick−plan−start;
        !negotation;
        pick−plan−rest.
+!negotiation: request(Sender, Msg) ←
        negotiation−plan.
```

Obviously, this technique leads to implementation of agent behaviours in terms of finite state machines and consequently to brittle, non-elaboration-tolerant, code. In order to add a new behaviour, interactions with all the other existing behaviours have to be considered and these have to be modified accordingly.

An alternative solution supported by the *Jason* interpreter is to employ .suspend and .resume internal functions which facilitate suspension and resuming of intentions respectively. The previous example could then be reformulated as follows:

```
+!formation_alignment : context ←
        align−plan.
+!pick_object : see(Object) ←
        .suspend(formation_alignment);
        pick−plan;
        .resume(formation_alignment).
+!negotiation: request(Sender, Msg) ←
        .suspend(pick_object); suspend(formation_alignment);
        negotiation−plan;
        .resume(pick_object); .resume(formation_alignment).
```

The presented code should be considered in comparison with the previous example which involved explicit invocation of the possible higher-priority interruptions. In this case, the approach is to rather let the lower-priority plans to proceed freely, while the higher-priority behaviours should care for suspending

and resuming the possibly running lower-priority plans. Clearly, both solutions suffer from the same problems and lead to brittle code in which plans for various independent behaviours have to be informed and have to depend on each other for the program to execute correctly.

The only scalable and flexible mechanism for the problem of interacting plans is customization of the intention selection function $\mathcal{S}_{\mathcal{I}}$. The modified function would prioritise the intentions appropriately according to the particular application domain. The downside of this, rather heavyweight, solution is that it renders the resulting *Jason* program to be not unambiguously readable and understandable in isolation. An important part of the program semantics is this way shifted to the *Java* side and the *Jason* program cannot be fully comprehended without understanding the *Java* code functionality.

Finally, in [8] authors discuss the plan annotation `priority` reserved for future use. The annotation is intended to instruct the plan selection and intention selection functions $\mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{I}}$ about the plan and intention selection priority respectively. They also note that the mechanism is not implemented in *Jason* programming system yet and do not provide enough technical detail on its functionality.

Above, we tried to show that the problem of steering plan interactions and interruptions is an important one, yet not solved appropriately in the current incarnation of *Jason*. On one hand, an intuitive and clean mechanism for intention interaction is vital in BDI-style agent programming, where several intentions might be running in parallel and interleave their executions. On the other, intentions can interact in many different ways. To strike balance between the two requirements, as an informal attempt, we suggest a conservative extension of *Jason* allowing to impose partial ordering of plans and intentions in a program. While certainly not a mechanism general enough (consider e.g., a specification of the priorities of the program modules, similar to the one proposed in [15]), such a mechanism, would help to avoid customisation of the intention selection function $\mathcal{S}_{\mathcal{I}}$, which we consider a bad design practice for the reasons discussed above.

4.3 Mental notes and plan destructors

Mental notes are beliefs added to an agent's belief base from inside its intention. This way the agent can remind itself about status of its own execution and partially solve the problem of intention interactions discussed in the previous section. The main reason to employ mental notes is to provide a way to transfer complex information between two behaviours, usually between a behaviour and its invoked subgoals. As a result, the mental notes can be used as a kind of local variables of plans. The belief base may have to be cleaned up upon an intention completion by retracting these "local variables" corresponding to the intention. If implemented carefully, *Jason* provides a means to implement such a mechanism. Consider the following code:

```
+!event: context ←
        +event(note1);
```

```
        ...;
        +event(note2);
        ...;
        .abolish(event(_)).
```

Each mental note local to the intention triggered by the event event is of a particular form, allowing a bulk retract of all the beliefs of one argument and name event using the internal action .abolish.

While relatively straightforward, this technique can lead to difficulties in the case of an intention failure. The involved problems are quite similar to those involved in handling run-time exceptions in imperative programming languages. Upon an intention failure, the local mental notes have to be cleaned up as well. A variation of the following can be used to achieve that:

```
−!event: context ←
        ...;
        .abolish(event(_)).
```

Besides code duplication, a naïve *Jason* programmer can simply forget to implement the appropriate failure plan. Another issue of this technique is that it might be necessary to use a different mental note forms for alternative plans handling the event event. However, upon an intention failure it is no longer possible to recognise, which particular intention, has failed.

We informally propose a language extension similar to the exception handling programming construct try−catch−finally present in many imperative languages, as well as in some niche agent programming languages, such as *StorySpeak* [12]. Consider the following code snippet:

```
+!event: context ←
        try {
                plan−body;
        } finally {
                .abolish(event(_));
        }.
```

The code in the finally block should include a plan destructor, a subplan which should be invoked upon the plan termination, regardless of its success, or a failure. The advantage of this construct is that the plan destructor is associated with the particular plan variant handling event +!event, unlike the standard *Jason* plan failure event −!event. Obviously the syntax of the proposed extension is is not in line with the declarative spirit of *AgentSpeak(L)* and *Jason*, but it illustrates the point well.

4.4 Jason agents vs. external environment

In the implemented case-study, agents had a time limit imposed on their deliberation. They had 2500ms to choose their next action. If the action is not chosen within this timeframe, the simulated environment continues as if the agent executed the action skip and discards any action reply delivered after the timeout. In such environments, it is vital for an agent programmer, to optimise and speed up the agent's deliberation as much as possible. However, speeding up the deliberation itself is often not enough. The agent may then have to restart a whole

intention (or just a single instantiated plan) to take new state of the environment into account.

In the implemented case-study, it was necessary for agents to reason about complex aspects of the environment, such as relative positions of teammates in a formation. In order to speed up the deliberation of the agent, we have implemented a relatively complex mechanism of belief updating. Upon each belief update, an agent triggers an event for a plan pre-calculating answers to often-queried context conditions and storing them as mental notes in his belief base. While speeding up the execution, this mechanism led to relatively complex belief base handling within the agent. However, even with this optimisation, the agents were sometimes not able to reply to the server within the set time limit in some situations.

To solve the problem of a prolonged deliberation, we propose two extensions of the *Jason* programming system. Prolonged reasoning over the agent's beliefs is often invoked from the rule context conditions (deliberation over complex aspects of the environment, such as the form of obstacles ahead, path calculation, etc.). In order to speed up such *Prolog* query evaluations, we propose to implement a RETE-style mechanism [11] for context conditions which can be calculated once and then treated as constant queries for the rest of the deliberation cycle.

To deal with the intention restart problem, *Jason* provides constructs for explicit management of the intention base. Current implementation of the *Jason* programming system provides the internal action .drop_intention facilitating forceful intention cancellation from within a plan of the agent. The straightforward use of this mechanism is however not well suited for the case-study application. It would require implementation of a recurring goal, a loop like pattern, regularly checking whether the timeout already passed, or not. Another option would be to add the timestep mechanism handling to the environment implementation, annotate the relevant plans with a particular name pattern and finally enhance the agent program with a plan similar to the following one:

```
+timestep: true ←
        .drop_intention(...);
        /* possibly restart some of the intentions */.
```

Usage of design solutions such as the two introduced in the previous paragraph, however, would interact with other plans as discussed in Subsection 4.2 and would be difficult without an appropriate customisation of the intention selection function. Secondly, and perhaps more importantly in the case of the first solution, regularly checking the timeout could lead to further slow-down of the deliberation cycle.

We propose an extension of the *Jason* annotation mechanism allowing annotation of the agent's intentions with timestamps. At the point when the system timestamp value is incremented, either by the agent program itself, or from within the underlying *Java* code, all the intentions annotated with a lower timestamp should automatically fail as they become irrelevant.

To conclude this part, let's consider interaction between the *Jason* interpreter and an external environment in general. In its current incarnation, *Jason* is rather *introverted*, as are many other agent-oriented programming languages. In

particular, the programming system implicitly assumes that the agent acts in a synchronous manner with respect to the environment. This assumption holds when the speed of the agent's deliberation is higher, or at least matching the rate of change, the update frequency, of the environment. However, in cases where the agent deliberation struggles to match the frequency imposed by the environment, the current implementation of the *Jason* programming system does not provide enough optimisation mechanisms to deal with the issue (in [13], we discuss some possibilities dealing with this problem in the context of videogame bots).

4.5 Jason vs. Java

Jason programming system is tightly integrated with the underlying *Java* environment. This setup allows interfacing the implemented agents with their environments in a very flexible way. It also facilitates extensive customisation of the language interpreter for the particular application domain. *Jason* allows for custom belief bases, as well as adaptation of the event, plan and intention selection functions $\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{I}}$ respectively.

We argue, that the flexibility of this setup might also be a drawback. The reason is that such extensive customisations may lead to a fuzzy boundary between *Java* and *Jason* parts of the implemented agent program. Significant and important parts of the agent program functionality are often implemented in *Java* code, but this approach tends to render the *Jason* (*AgentSpeak(L)*) program difficult to understand in isolation.

In this context, a point especially relevant for novice *Jason* programmers, is the question *what are the guidelines regarding which aspects of the agent program should be implemented in Java and which in Jason?* In an extreme case, one could consider a trivial *Jason* program of the following form:

```
!main.
+!main: true ← .main.
```

There is a single event invoked at the start of the program, which leads to an invocation of an internal action `main` implementing the whole functionality of the agent as a *Java* code. In contrast to this approach, we may have the A* search algorithm implemented exclusively in *Jason*. While both of these *Jason* programs are absurd, they illustrate the point.

The ability to shift pieces of functionality between *Java* and *Jason*, and at the same time not having clear guidelines regarding what belongs where, leads to confusion of inexperienced *Jason* programmers. Bordini, Hübner and Wooldridge briefly mention this issue in [8], Chapter 11. They seem to take a puristic stance, since they argue that programmers should resist the temptation to enhance environments with "fake" actions and other user customisations leading to "cheating" in *Jason* programming. While this point is fair, the pragmatic use of the *Jason* language by a relatively inexperienced programmer facing design issues such as those discussed above in this section might lead to a growing frustration and finally a solution through the path of "minimal effort". The programmer might simply revert to a more familiar tool, in this case the *Java* programming language.

A similar issue has been addressed by J. J. Bryson in the context of the *POSH* reactive planner [10]. She proposes a methodology for a behavioural design, which, besides other things, states explicitly, which parts of behavioural code belong to underlying Java (or Python) and which to *POSH*.

4.6 Minor technical and methodological issues

Finally, let us conclude the core discourse by listing some minor technical issues a programmer learning the *Jason* programming system encounters. While of relatively low importance, improvement on these fronts could have an impact on overall usability of the *Jason* programming system.

Debugging Debugging BDI agent systems is a topic often discussed within the community. Apart from deeper discussion on particular debugging methods, one of the issues are the appropriate tools available for the particular programming platform. *Jason* provides a tool for stepping through the agent's reasoning cycle, display its current belief base, the pursued intentions and events awaiting evaluation. Apart from problems with stability of the tool, one of the main difficulties with this style of program debugging is that in situations with relatively short time limit on agent's deliberation, this approach is inapplicable. A more appropriate technique in such situations is to use a logging facility.

In *Jason* ver. 1.3.3, which has been used for this study, the provided logger does not expose enough information to the programmer. It is not comprehensible enough, as, apart from user defined outputs, it only reports selected events and plans, percepts and execution control messages. It would be useful to dump/export the whole current state of the agent when needed. Additionally, the user should be allowed to specify different levels of detail for logging (even dynamically during the execution), as output of whole states could be sometimes space intensive.

Integrated Development Environment Even though the provided *Eclipse* plug-in is reasonably comfortable, it does not follow some of the established patterns for plug-ins of the same category for *Eclipse* IDE. Instead of adding program run options directly to the project options menu, it has them attached to the context menu of a *mas2j* file. An ordinary *Eclipse* plug-in would try and replicate the selection of main class of *Java* program, which has essentially the same objectives.

Another minor issue is the lack of code completion function in the standard *Jason* IDE, which rather slows down agent program implementation.

Educational material One of the most difficult aspects of programming in *Jason* was actually learning it. There is only a limited material freely available. Thus, along with generated documentation for the source code (*javadoc*),

examples and demos, the most useful resource is the book *"Programming Multi-Agent Systems in AgentSpeak Using Jason"* [8]. While the book provides a complete description of the programming system itself, it is still relatively difficult to use as a pedagogical tool. It imposes a strong emphasis on the theoretical part of *Jason*, without introducing the student into pragmatics of building more complex agent systems first. To improve the situation, we would appreciate several authoritative tutorials on incremental building of complex agent systems teaching the correct techniques of programming in *Jason*. As of now, the initial barrier between first working plans and first complex interacting plans is tremendous and requires a lot of trial and error approach on the side of the novice *Jason* programmer. In our opinion, the hurdle is much greater than those of other, especially imperative, languages such as *Java*, *C++* or *Python*.

5 Final remarks

In the above sections, we discussed some of the most problematic issues we have faced during the experiment. In particular, the experiment aimed at an implementation of a relatively complex case-study application by a programmer without a prior knowledge of *Jason* language. To keep the experience as relevant to *Jason*-style agent programming as possible, one of the goals was to try to use *Jason* programming system as is, with as few customisations as possible. In particular, we decided not to customise the deliberation cycle of the *Jason* interpreter and to limit the code written in *Java*. The features implemented in a form of a *Java* code were those facilitating the interaction with the simulated environment, such as a set of internal actions implementing path planning algorithms.

Since we have used the *Cows & Cowboys* simulated environment for the *Multi-Agent Programming Contest* (*AgentContest*), the complexity of the implemented case-study is directly comparable to the implementations of *AgentContest* entries in its last few editions. For comparison, our implementation resulted in a codebase involving 1127 lines of code, while the *AgentContest* entries to editions 2009 and 2010, presented by teams involving the Jason platform developers, included 1416 and 1648 lines of code respectively. The *AgentContest* entries, however, aimed at the full-featured cows herding scenario, while our case-study implemented only a fragment of the scenario, environment exploration and movement in a formation through the environment. The independent entry to the 2010 edition of the *AgentContest* by the team of the *Technical University of Denmark* featured only 173 lines of *Jason* code and most of the team functionality was thus implemented on *Java* side. If our assumption that the *AgentContest* entries are the largest publicly available applications written to date is correct, then our case-study resulted in one of the most extensive *Jason* codebases to date.

In parallel to creating the here reported *Jason* implementation, several students implemented the same case-study application in *Java* in the context of Multi-Agent Systems course at CTU in Prague. Interestingly, while most of them considered the task quite work-intensive and reported a workload in range

of 40-60 hours of programming and testing to complete the undertaking, the *Jason* implementation took more than 100 hours to complete for an experienced *Java* programmer. The average *Java* codebase resulting from the exercise involved more than 4000 lines of code. While no hard conclusion can be drawn from this remark, it can serve as an indicator that learning *Jason* on a non-trivial example application is definitely a difficult task and that the community should invest more effort into educational material such as more extensive tutorials on teaching of agent-oriented programming.

The discussion in this paper does not aim at providing a significant scientific contribution. However, we believe that reports, such as this, contribute to the on-going discussion in the community on usefulness, relevance and pragmatics of agent-oriented programming systems, tools and languages, as well as to the future developments of the field. We would like to emphasize that the issues discussed in this paper are those we found to be important while developing a concrete experimental case-study. The conclusions drawn here, even if generic as they are, should be considered with caution in the context of the particular application domain. To study the subject in a more depth and more rigorously, further studies on larger groups of test subjects should take place and should also consider some established methodologies like Agent-Oriented Software Engineering.

Acknowledgements We are grateful to Jomi F. Hübner (*Federal University of Santa Catarina*, Brasil) and Jørgen Villadsen (*Technical University of Denmark*) for the permission to study and use the code of their entries to the AgentContest.

Authors of the presented work were supported by the *Czech Ministry of Education* grants MSM6840770038 and MSM0021620838; the *Grant Agency of the Czech Technical University in Prague* grant SGS10/189/OHK3/2T/13; the *Grant Agency of Czech Republic* grant P103/10/1287 and the *Grant Agency of Charles University in Prague* grant 0449/2010/A-INF/MFF.

References

1. Multi-agent programming contest 2010. http://www.multiagentcontest.org/2010, 2010.
2. Tristan M. Behrens, Mehdi Dastani, Jürgen Dix, Michael Köster, and Peter Novák. The multi-agent programming contest from 2005-2010 - from gold collecting to herding cows. *Ann. Math. Artif. Intell.*, 59(3-4):277–311, 2010.
3. Tristan M. Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. MASSim: Technical Infrastructure for AgentContest Competition Series. http://www.multiagentcontest.org/, 2009.
4. Tristan Marc Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. Multi-Agent Programming Contest. http://www.multiagentcontest.org/, 2009.
5. Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

6. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
7. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.
8. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley-Blackwell, 2007.
9. Michael E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999.
10. Joanna J. Bryson. *Intelligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent*. PhD thesis, 2001.
11. Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.
12. Jakub Gemrot. Joint behaviour for virtual humans. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2009.
13. Jakub Gemrot, Cyril Brom, and Tomáš Plch. A periphery of pogamut: From bots to agents and back again. In Frank Dignum, editor, *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, volume 6525 of *Lecture Notes in Computer Science*, pages 19–37. Springer Verlag, 2011.
14. Jason Developers. Jason, a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net/, 2011.
15. Neil Madden and Brian Logan. Modularity and Compositionality in Jason. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *PROMAS*, volume 5919 of *Lecture Notes in Computer Science*, pages 237–253. Springer, 2009.
16. Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations. In Koen V. Hindriks, Alexander Pokahr, and Sebastian Sardiña, editors, *ProMAS*, volume 5442 of *Lecture Notes in Computer Science*, pages 72–87. Springer, 2008.
17. Peter Novák and Wojciech Jamroga. Code patterns for agent-oriented programming. In *Proceedings of The Eighth International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS*. IFAAMAS, 2009.
18. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter 6, pages 149–174. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [7], 2005.
19. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Walter Van de Velde and John W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.
20. Michael Winikoff. *JACK^{TM} Intelligent Agents: An Industrial Strength Platform*, chapter 7, pages 175–193. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [7], 2005.