Notes on pragmatic agent-programming with Jason

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

[1] Department of Software and Computer Science Education
Faculty of Mathematics and Physics, Charles University in Prague
Czech Republic

[2] Agent Technology Center, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague
Czech Republic

Abstract *AgentSpeak(L)*, together with its implementation *Jason*, are 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 work, we present our experiences gathered during an experiment aiming 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 aimed at solving the discussed design problems and programming language issues.

1 Introduction

Jason [6] is an agent-oriented programming system implementing the agent programming language *AgentSpeak(L)* [14]. *AgentSpeak(L)* was proposed as a theoretical language, an articulation and operationalization of the Bratman's Belief-Desire-Intention architecture [7]. *Jason* is nowadays one of the most prominent approaches in the group of theoretically-rooted agent-oriented programming languages (APLs). 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, e.g., *2APL*, *3APL*, *GOAL*, *Jazzyk*, *Golog* (for

an overview consult e.g., [3,5,4]) 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. [13,15]), these languages, including *AgentSpeak(L)*, were constructed from scratch which also led to serious shortcomings with respect to practicality of their use.

While on one hand, pragmatic problems of agent design and implementation, such as e.g., code modularity, are gaining a more prominent role in the research community, on the other, the 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 in the research community on its practical applications, be it more involved applied research projects, or more significant close-to-real-world applications (cf. also the *Jason* related projects website [11]). The only report on pragmatic issues faced when using *Jason* in a more involved context is the recent study by Madden and Logan [12] in which the authors deal with problems of modularity in their application and in turn propose a 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 8 submissions in years 2006-2010 altogether by three independent research groups. The reports on development of these applications, however, do not include discussion of the practical issues of agent program implementation, but rather focus on the analysis and design issues with an emphasis on the multi-agent coordination.

In this paper we discuss our experiences gathered during an experiment aiming at development of a non-trivial case-study agent application by a novice *Jason* programmer. The main goal of the undertaking was exploration of basic problems in multi-agent coordination in a simple simulated environment using the *Jason* programming system. In particular, we implemented an application involving a team of 8 agents collaboratively exploring a grid maze and subsequently traversing the environment while cooperatively maintaining a formation. Our experiment aimed at a naive, relatively conservative, use of the *Jason* programming system. I.e., 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 [11] 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*, so that it will be useful for the wider community involved in the research on agent-oriented programming languages and tools. 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 into *AgentSpeak(L)* and *Jason* in Section 2 and the description of the implemented case-study (Section 3), in Section 4, the core of

this paper, we discuss a selection of problems we 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 subsequently we discuss the possible solutions. The topics covered in the discussion include implementation of a simple loop design pattern, handling of interactions between several plans and interruptibility thereof, usage of mental notes as local variables in plans and two technical issues arising from implementation of agents embodied in dynamic environment and the unclear boundary between *Jason* programming language itself and its underlying extension/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 [14]. 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 the *AgentSpeak(L)* with various extensions and 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, namely

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 which can be implemented as *Java* methods. Furthermore, *AgentSpeak(L)* disregards the implementation details of agent's interaction with its environment. In particular, 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 in the belief base and also provides a toolbox for implementation of custom belief bases meant as a means for representing complex beliefs, such as the topology of environments, or interface to relational databases. Finally, *Jason* provides a framework for 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 a 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 with minimal customisations 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. [2], scenarios for the 2009-10 editions) is a challenging scenario for benchmarking cooperative multi-agent teams. 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 size approximately 100 cells wide. 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. Agents can move between empty cells and are able to open fences located in the environment, by pushing a button at the edge of each fence. Similarly to agents, cows are also able to move between empty cells, however their movement is steered by the environment and takes into account their mutual distances, as well as the distances from the agents and the trees the cow can see. Agents and cows have a limited view, and in each simulation step receive a perception containing cells in their vicinity (agents see a square of 17×17 cells centred at the agent's position, cows see a square of 11×11 cells). The task of each agent team is to herd as many cows as possible into a corral belonging to the team. As cows are afraid when they see an agent, they can be pushed by a coordinated team of agents in a particular direction.

For the purposes of the here reported case-study, we 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

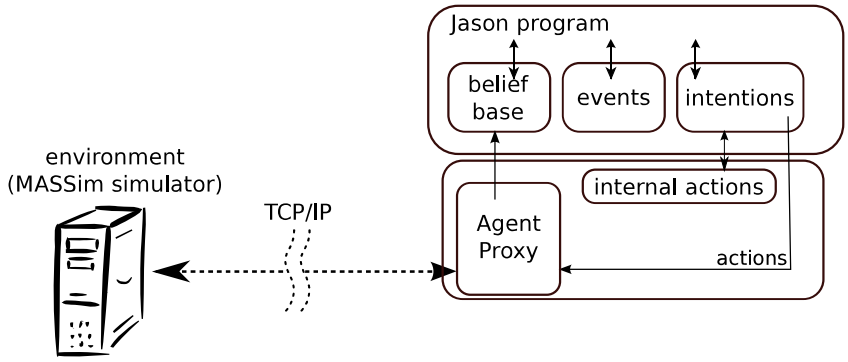

Figure 1. The scheme of the architecture of a single *Jason* agent interacting with the simulated environment.

landmarks and then traverse the maze from one landmark to another while maintaining a formation of a particular shape.

The simulated environment was provided by the MASSim server [1]. The scheme of the implemented system is depicted in Figure 1. The interaction with the simulator was implemented in *Java* by the class AgentProxy derived from the AbstractAgent example stub class provided together with the MASSim package. In each simulation step, the MASSim server sends to each agent an XML message encoding the agent's perception, essentially the content of the cells the agent sees. VisionProcessor, a component of the AgentProxy object, then decodes the XML message and updates the belief base of the agent accordingly.

Upon an update of the belief base, the *Jason* interpreter triggers a set of belief update events, which serve to maintain up-to-date state and consistency of the belief base, as well as to pre-compute answers to some often requested and at the same time computationally-intensive queries. The belief base itself is customised to support unique beliefs. Its implementation replicates the belief base from Gold Miners II example from the standard distribution of *Jason*. Unique beliefs are agent's location, its team mates' position, timestep and similar.

Following the updates of the belief base, an action from the previous timestep, if there was a one, is marked as executed and *Jason* thread continues its deliberation upon new percepts. The AgentProxy control thread then 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 some intention of the agent. The AgentProxy then validates the action (correct timestep), and if it is valid, it is then translated into the corresponding XML message and 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 the lottery-like mechanism for choosing the formation leader, queries for contents of map cells, etc.

One of the most important constraints on the implementation of the case-study was that we do not customise the *Jason* interpreter itself, neither the event, plan and 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 by a programmer learning the *Jason* language along the way. As the authoritative source and documentation, the book *Programming multi-agent systems in AgentSpeak using Jason* [6] was used. For clarity, the discussion of each issue includes a brief motivation and explanation of the particular design problem, subsequently 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

Often a designer needs to implement some kind of a loop design pattern. E.g., in a maze-like environment, the agent calculates a path from a point A to a point B using some path planning algorithm and then it should follow the path. This design 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
```

Jason does not feature a loop programming construct[3], however it can be implemented by the following *Jason* code:

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

This design implements the idea of tail recursion. However, as in the current versions of *Jason*, the interpreter does not feature a special treatment of tail recursion, according to the language semantics, this design unfolds into an ever growing intention stack. At the bottom of the stack is the plan `after−loop−plan` with a series of invocations of `!loop` of length equal the number of iterations of the loop. In order to facilitate correct plan failure handling, *Jason* interpreter

[3] From the version 1.3.4 *Jason* interpreter includes an explicit loop programming construct. The update was however released only after finishing the here described experiment and the authoritative source on *Jason* [6] does not discuss this issue.

does not remove top-level event 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. In cases with extremely high number of loop iterations, e.g., several dozen thousands path steps is not that extreme for large grid environments, the intention stack growth can lead to high memory consumption, and perhaps more importantly, upon reaching the loop condition, prolonged intention stack cleanup before the interpreter continues with the after−loop−plan.

A naive attempt by a novice programmer could be a loop implementation using the asynchronous event invocation !!loop, straightforward use of which however is inappropriate in this context as besides invoking the loop, it would lead to 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. [6], 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, i.e., 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 intention stack length increase, while at the same time still allows for plan failure handling as in the standard loop implementation.

In the pattern above, the loop recurring event carries with it the appropriate callback to invoke 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.
```

Loop is a handy and often used design pattern. However, for a novice programmer, loop implementation in *Jason* is rather unintuitive and its implementation often leads to a confusion. One of the straightforward solutions, well in the spirit of BDI architecture, would be use of persistent goals, such as in 3APL. Another way to deal with this confusion 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 [6], Chapter 8.

4.2 Interruptions and plan 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 plans, which can mutually interrupt each other. In *Jason*, similarly to most state-of-the-art BDI-based agent programming languages, plans are considered implicitly interruptible. However, having several plans 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?*.

There are basically three solutions to this problem. The straightforward solution 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 shows use of the 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 plans. 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 plans would be to let the program handle the situations, in which they can be interrupted, not the plans themselves. I.e., all plans would be considered implicitly non-interruptible and at every point when a plan can be interrupted by a higher-priority event, there would be an explicit check for all the possibilities of such interruptions, a synchronous invocation of the interrupting event, followed by an explicit check for preconditions of the remaining plan. The following code snippet demonstrates use 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.

The only scalable and flexible mechanism solving the problem of interacting plans is the customisation of the intention selection function $\mathcal{S}_{\mathcal{I}}$ in *Java* so that it prioritises the running 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 [6], 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, resp. intention selection priority. However, 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 plan interaction is vital in BDI-style agent programming, where several plans might be running in parallel and interleave their executions. On the other, plans can interact in too many different ways. To strike balance between the two requirements, as an informal attempt, we call for a conservative extension of *Jason* allowing to impose partial ordering of plans in a program. While certainly not a mechanism general enough (consider e.g., specification of the priority of program modules, similar to the one proposed in [12]), such a mechanism would, in many cases, help avoid customisation of the intention selection function $\mathcal{S}_{\mathcal{I}}$, which we consider a bad design for the reasons discussed above.

4.3 Mental notes and plan destructors

Mental notes are a means for an agent to modify its belief base from its plans in run time. This way the agent can remind itself about status of its own execution and thus partially treat the above discussed problem of plan interactions. Another use of mental notes is transfer of complex information between a behaviour and its invoked subgoals. In result, the mental notes can be used as a kind of local variables of plans. Often, after plan completion, the belief base should be cleaned up, i.e, a programmer would like to retract the set of "local variables" corresponding to the plan. If implemented carefully, *Jason* provides a means to implement such a mechanism. Consider the following code:

```
+!eventX: context ←
        +eventX(note1);
        ...;
        +eventX(note2);
        ...;
        −eventX(_).
```

I.e., each mental note local to the plan triggered by the event eventX is of a particular form, allowing later a bulk retract of all the beliefs in that form from the belief base.

While relatively straightforward, this technique can lead to difficulties in the case of plan failure. Firstly, upon plan failure the local mental notes have to be cleaned up as well, i.e., always when using such mental notes, a plan failure code similar to the following should be used:

```
−!eventX: context ←
        ...;
        −eventX(_).
```

Besides code duplication, a novice *Jason* programmer can simply forget to implement the appropriate plan failure mechanism. Another problem of this technique is that it might be necessary to use different mental note forms for alternative plans handling the event eventX. However, upon plan failure, it is no

longer possible to recognise which particular plan handling the event failed, what can lead to difficulties with the belief base clean up.

We informally propose a language extension similar to exception handling programming construct try−catch−finally present in many imperative languages, as well as in some niche agent programming languages, such as e.g., *StorySpeak* [9]. Consider the following code snippet:

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

In the finally block, code includes a plan destructor, i.e., a subplan which should be invoked upon 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 the event unlike when using the standard *Jason* plan failure event invocation −!event .

4.4 Jason agents vs. external environment

In the implemented case-study, the agents had a time limit imposed on the length of their deliberation. In particular, they were allowed 2500ms to decide upon their next actions. If the action was not issued within the timeout, the simulated environment went on as if the agent executed the action skip and discarded any action reply delivered after the timeout. In such environments, it is vital for the agent programmer firstly, to be able to optimise and speed up the agent's deliberation as much as possible, and secondly, to be able to steer the deliberation cycle of the agent from within its plans.

In the implemented case-study, it was necessary for the agent to reason about complex aspects of the environment, such as the form of obstacle structures ahead, fence structures, etc. In order to speed up the deliberation of the agent, we implemented a relatively complex mechanism of belief updating. Upon each belief update, the agent triggered an event to pre-calculate answers to often-queried context conditions and stored them as mental notes in its belief base. While serving the solution, this mechanism led to relatively complex belief base handling within the agent. However, even with this optimisation, it often happened that the implemented agents were not able to reply to the server within the set time limit.

To solve this problem, we propose two extensions of the *Jason* programming system. Prolonged reasoning over the agent's beliefs is often invoked from the rule context conditions (e.g., 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 [8] for context conditions which can be calculated only once and treated as constant queries for the rest of the deliberation cycle. In result, we propose introduction of annotations of rules, or their context conditions, with a

flag denoting their constant value throughout a deliberation cycle, or even until a special belief update event is triggered.

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 afresh */
```

The invocation of the action .drop_intention(...) drops all intentions matching the pattern provided as the argument.

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 to include a possibility to annotate agent's intentions with integer values, timesteps. At the point when the system timestep value is incremented, either by the agent program itself, or from within the underlying *Java* code interfacing the agent with the environment, all the intentions annotated with lower timestep value should automatically fail, because they are no longer relevant.

To conclude this part, in its current incarnation, *Jason*, similarly to many other agent-oriented programming languages, is rather *introverted*. 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 relatively higher, or at least matching the rate of change, resp. speed of update, of the environment. However, in cases when the agent deliberation struggles to match the rate of change imposed by the environment, the current implementation of the *Jason* programming system does not provide enough optimisation mechanisms to deal with the issue (we discuss possibilities to deal with this problem in the context of videogame bots in [10]).

4.5 Jason vs. Java

As already remarked above, *Jason* programming system is tightly integrated with the underlying *Java* environment. This setup allows interfacing the implemented agents with their environments in a flexible way, as well as it provides great possibilities with respect to customisation of the language interpreter for

the particular application domain in terms of custom belief bases, and specially tailored 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, however, is also a major drawback. As already discussed above, such customisations lead to an unclear boundary between *Java* and *Jason* parts of the implemented agent program. Often, significant and important parts of the agent program functionality are implemented in *Java* code what renders the *Jason* program itself only hardly understandable in isolation.

Another 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, this might lead to a trivial Jason program of the form:

```
!main.
+!main: true ← .main.
```

I.e., there is a single event invoked at the start of the program which leads to invocation of an internal action `main` implementing the whole functionality of the agent as a *Java* code. While such a *Jason* program is absurd, it illustrates the point. The possibility 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 programmers.

Bordini, Hübner and Wooldridge touch on this issue in [6], 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 the point is fair, pragmatic use of the *Jason* language by a relatively inexperienced programmer facing design issues such as those discussed in this section might lead to a series of implementation stages characterised by a growing frustration with the programming system concluded by an escape to the path of "minimal effort", i.e., using a more familiar tool, in this case *Java* programming language.

4.6 Minor technical and methodological issues

Finally, let us conclude the core discourse by listing of some minor issues a programmer learning the *Jason* programming system encounters.

Debugging Debugging of BDI agent systems is a problem known and discussed in the community. Apart from deeper discussion on particular debugging methods, one of the issues are the debugging tools available within 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 unusable. A more appropriate technique in such situations is to use a logging facility.

However, in the *Jason* implementation ver. 1.3.3, which has been used for this study, the provided logger does not provide enough information for 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 export the whole current state of the agent, provided the user is allowed to specify different levels of detail for logging (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 run options directly to the project options, 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 objective.

Another issues 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 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"* [6]. While the book provides a complete description of the programming system itself, it is still relatively difficult to use it 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. To improve the situation, availability of several authoritative tutorials on incremental building complex agent systems would definitely help to promote the correct programming techniques in the language. 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, it is much greater than for other languages, such as *Java*, *C++* or *Python*.

5 Final remarks

In the above sections, we discussed some of the most problematic issues we faced during the experiment. In particular, the experiment aimed at 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 strict prior requirements was to try to use *Jason* programming system as is, i.e., with as few customisations as possible. In particular, we decided not to customise the deliberation cycle of the *Jason* interpreter and the only parts implemented as *Java* code were those facilitating the interaction with the simulated environment, i.e., a set of internal actions

implementing e.g., path planning algorithms and some arithmetic calculations and the customised belief base handling the availability of perceptions received from the simulated environment from within *Jason* code.

Since we used the simulated environment provided in 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, which featured the *Cows & Cowboys* scenario, i.e., the same simulated environment. For comparison, our implementation resulted in 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 conducting 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 120 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 hard task and the community should also invest more effort in promoting educational material and more extensive tutorials on teaching 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.

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* 0449/2010/A-INF/MFF. Preparing this text and presenting this work was also sup-

ported by the project CZ.2.17/3.1.00/31162 that is financed by the *European Social Fund* and the *Budget of the Municipality of Prague* (for Radek Pibil).

References

1. Tristan M. Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. MASSim: Technical Infrastructure for AgentContest Competition Series. http://www.multiagentcontest.org/, 2009.
2. Tristan Marc Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. Multi-Agent Programming Contest. http://www.multiagentcontest.org/, 2009.
3. 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.
4. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
5. 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.
6. 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.
7. Michael E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999.
8. Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.
9. Jakub Gemrot. Joint behaviour for virtual humans. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2009.
10. 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.
11. Jason Developers. Jason, a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net/, 2011.
12. 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.
13. 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* [5], 2005.
14. 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.
15. 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* [5], 2005.