Pitfalls of Jason Concurrency

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

Grupo Babel, DLSIIS, ETSIINF, Universidad Politécnica de Madrid
alvarofdezdiaz@gmail.com cbenac@fi.upm.es lfredlund@fi.upm.es

Abstract

Jason is a well-known programming language for multiagent systems where fine-grained concurrency primitives allow a highly-concurrent efficient execution. However, typical concurrency errors such as race conditions are hard to avoid. In this paper, we analyze a number of such potential pitfalls of the Jason concurrency model, and, describe both how such risks can be mitigated in Jason itself, as well as discussing the alternatives implemented in eJason, an experimental extension of Jason with support for distribution and fault tolerance. In some cases, we propose changes in the standard Jason sematics.

1 Introduction

Jason [1] is a well-known programming language for multiagent systems (MASs) with a well established formal semantics based on Agent Speak [6].

Programming multiagent systems is not an easy task, as it involves coordinating the concurrent execution of a set of independent agents (akin to processes in mainstream concurrent programming), each of which may also be composed of a set of independently executing intentions (in mainstream programming often named threads). In Jason agents communicate through message passing, afording a high-level of control, whereas intra-agent communication between intentions is realised through asserting and retracting beliefs in the shared belief base.

In previous work we have introduced eJason [3, 4], an extension to Jason where new features have been added to cope with distribution and fault-tolerance in MAS. As a first step to extend Jason with these features, we analyze in this paper to what respect the concurrency model of Jason enables programmers to develop concurrent multiagent systems without running into the usual pitfalls of concurrency, i.e., difficult to handle race conditions. Clearly the fine-grained concurrency primitives present in Jason, where the belief database is shared among all concurrent interactions, promises highly-concurrent efficient execution, but at the same time the programmer should be provided with convenient high-level language constructs for controlling the amount inter-agent concurrency.

Intra-agent concurrency has been an issue of study for some time, in particular regarding how to resolve conflicts among goals when agents pursue multiple goals. In [7] conflicts are handled in the goal level by representing conflicting

goals. A difficulty with this approach is that all plans for a conflicting goal are considered conflicting, i.e., non-conflicting alternative plans that can achieve the same goal are not considered. This issue is addressed in [8] which examines different strategies for resolving conflicts, such as dropping intentions, or modifying intentions with regards to the selection of plans for solving goals. Automatic detection of conflicts is the approach followed by [9], where means for reasoning about the goal interactions are incorporated into the commercial BDI agent development platform JACK, and evaluated empirically.

In this paper we take Jason as the programming platform of study and discuss its semantics and implementation with regards to some concurrency problems that may arise.

The rest of the paper is structured as four sections which each describe a potential difficulty with a Jason concurrency mechanism, discuss how the difficulties can be mitigated in Jason itself, and alternative solutions implemented in eJason. In Sect. 2 we discuss mechanisms to control the amount of concurrency among a set of interaction, whereas. Sect. 3 examines the possibility that the context of a plan may be false when the plan body starts executing. Next, in Sect. 4 and Sect. 5 we consider the mechanisms for handling failing achievement and test goals respectively, and alternatives to such early failures, i.e., goal suspension. Sect. 6 discussed how the changes implemented in eJason impacts the reasoning cycle of agents, whereas Sect. 7 draws a number of conclusion from the study of mechanisms to coordinate concurrent activities in Jason and eJason.

2 Mechanisms for synchronizing access to shared beliefs

The interpreter of Jason allows each agent to possess several foci of attention, corresponding to the different intentions of the agent. These intentions compete for the attention of the agent, and the decision on which intention to execute, in each iteration of the reasoning cycle, is determined by the agent's intention selection function. The execution order of the different intentions is not always irrelevant. The plans in the different intentions access and update the information stored in the agent's belief base. Therefore, the modification of the set of beliefs, derived from the execution of an intention, may affect the outcome (or even totally prevent the execution) of the rest of intentions available. The programmer must then consider these data dependencies between the different intentions of an agent's mental state and, when necessary, control the synchronisation of the execution of such intentions.

2.1 Nondeterministic execution implies nondeterministic belief bases

To illustrate the difficulties that may be caused by sharing beliefs among a set of interactions, consider the Jason agent in Fig. 1, which maintains a counter of the number of files that it has uploaded.

```
+!load(File) <-
   load(File);            // a
   ?files_loaded(Num);    // b
   -+files_loaded(Num+1). // c
```

Fig. 1. Jason plan for the file counter

Consider an agent with only this plan in its plan base and with initial goals g_1 = `!load(file1)` and g_2 = `!load(file2)`. The intentions corresponding to these goals are composed by one instance of the plan above, i.e. I_1=$[p_1]$ for g_1 and $I_2 = [p_2]$ for g_2 with plan bodies $\{a_1; b_1; c_1\}$ and $\{a_2; b_2; c_2\}$, respectively, where a_1 represents the formula a applied to g_1, and so on.

Several executions of the agent code, using the standard intention selection function, show that the counter is not always properly updated, as sometimes it only records the upload of one file, while, in fact, two files have been uploaded. A simple exploration of all possible execution traces, represented by the different possible interleavings of the actions in the plan bodies, exposes the root causes of the problem. These interleavings are depicted in Fig. 2. This figure shows a graph where each node, labelled $I_1 I_2 Ctr$, represents a different configuration of the agent's mental state such that $I_i \in \{a_i, b_i, c_i\} \cup X$ for $i \in \{1, 2\}$ corresponds to the action to be executed if the intention I_i gets selected by the intention selection function (the symbol X is used as a placeholder if the corresponding intention has been fully executed) and Ctr is the value of the counter (i.e. a belief `file_loaded(Ctr)`). For instance, the node $b_1 X1$ corresponds to a mental state such that the selection of the intention I_1 implies the execution of the formula b_1, the intention I_2 has been fully executed and the belief base contains a belief `files_loaded(1)`. Note then that the node $XX2$ corresponds to the outcome desired (i.e. the counter is properly updated), while the node $XX1$, shadowed in the graph, corresponds to an undesirable one (i.e. when the counter only records the upload of a single file, instead of two). The different edges in the graph represent the transition between mental states, and their label corresponds to the formula executed during that transition. For instance, the outgoing edge a_1 (resp. a_2) from the node $a_1 a_2 0$ to the node $b_1 a_2 0$ (resp. $a_1 b_2 0$) represents the transition triggered by the execution of the formula a_1=`load(file1)` (resp. a_2=`load(file2)`). The analysis of the execution traces shows that the undesired outcome occurs when the actions b_1 and b_2 (i.e. the actions where the value of the counter is read) have been executed without the execution of neither c_1 nor c_2 (i.e. the actions that update the counter) in-between (i.e. all traces containing the state $c_1 c_2 0$). The reason is that, in these cases, one of the intentions is handling outdated information regarding the counter and, therefore, the result is incorrect.

2.2 Jason solutions

In a sense, the problem is a standard one in concurrent programming, i.e., how to prohibit "bad" program executions where the concurrent execution of different

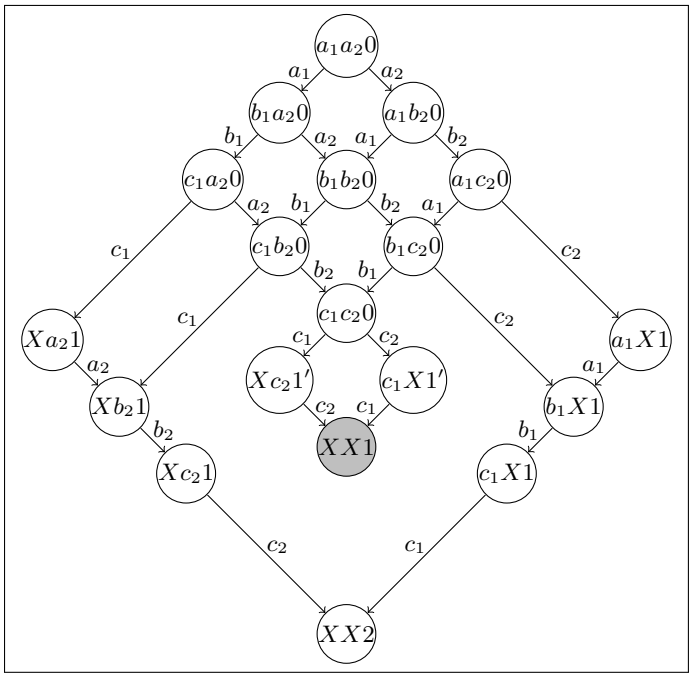

Fig. 2. Possible interleavings for the counter update example.

threads (or intentions in Jason) incorrectly interfere with each other due to concurrent access to a shared program state (in Jason, the belief base). The Java programming language, for instance, has synchronized objects to prevent concurrent access, and a number of more advanced mechanisms for controlling concurrent access available in the `java.util.concurrent` library.

Atomic Plans The Java implementation of Jason enables the labelling of plans as a way of including meta-level information that alters the agent's reasoning cycle. The label *atomic* is one of such labels. A plan labelled as atomic, also referred to as *atomic plan*, is such that, once this plan is selected for execution during an iteration of the reasoning cycle, all subsequent iterations will also select this intention until the atomic plan is fully executed. More informally, the atomic label represents a way of temporarily disabling the multiplicity of foci of attention, keeping the attention of the agent in the intention until the atomic plan is executed.

In order to avoid the data dependency explained in the previous section, an atomic plan can be used. For instance, by replacing the plan in Fig. 1 for the two (semantically dependent) plans provided in Figure 3.

This way, the formulas b and c are always executed consecutively, removing the execution traces that led to the wrong result. This solution can be used not only for belief updates, but also to implement behaviours that require the

```
+!load(File) <-
   load(File);            //a
   !update_counter.

@up[atomic]
+!update_counter <-
   ?files_loaded(Num);    //b
   -+files_loaded(Num+1). //c
```

Fig. 3. File counter with an atomic plan

agent to maintain its focus of attention on the same intention for several iterations of the reasoning cycle (i.e. executing the formulas from the same intention consecutively).

This solution requires the introduction of additional plans in order to determine the sets of actions that must be executed consecutively without a change in the focus of attention, even for relatively simple plans. In our opinion, this is not ideal from a programmer's perspective. It obscures the code, as it increases the number of plans in the agent's plan base, consequently increasing the complexity to maintain the agent's program. Moreover, in our opinion, providing several plans for the same goal shall be used to provide several alternatives for its accomplishment, not to implement continuations of the same plan.

2.3 eJason solution: Critical Sections

The eJason language proposed the definition of critical sections to reduce the amount of concurrency among interactions. Syntactically a critical section is enclosed within braces, i.e., "{{" and "}}". When an agent executes a formula within a critical section, there can be no concurrent change in the focus of attention as long as the critical section is not left (as happens during the execution of an atomic plan). Using critical sections as the synchronisation mechanism, the program for the agent that updates a counter, can be modified as shown in Fig. 4.

```
+!load(File) <-
   load(File);
   {{?files_loaded(Num);
   -+files_loaded(Num+1)}}.
```

Fig. 4. File counter with eJason critical sections

This plan provides the same functionality as the combination of the two plans in Fig. 3, i.e., Jason atomic plans and eJason critical sections are equivalent. In the case of atomic plans, the parametrisation (via labelling) of the intention selection function requires the programmer to consider the different intentions

that may conflict, and establish a priority order for their execution. In contrast, using critical sections, the programmer only identifies the regions of the agent's program that should be executed without interference from other intentions of the agent.

Labelling Conflicting Plans Recently Jason has been extended with a new interesting feature [10], where plans may be declared as *conflicting* with other plans, with the intention that conflicting plans may not be concurrently executed, whereas non-conflicting plans can be. In the example above, we can specify that the `up` plan conflicts with itself, by using the *conflict identifier* `self`. Then, the `[atomic]` declaration can be removed. Thus, in a sense, the possibility to explicitly label plans as conflicting refines the notion of atomic plans.

The ASTRA approach Another approach to controlling concurrency in languages based on AgentSpeak is demonstrated by the ASTRA [2] language. There critical sections are associated with an identifier (similar to the Java synchronized blocks), such that for any identifier, at any time there is at most one intention executing a critical section labelled by that identifier. Clearly, similarly to the approach with labelling conflicting plans, this proposal also permits an increased amount of concurrency (compared with using an universally shared critical section) among a set of concurrent interactions.

2.4 What is the right solution?

Providing programming languages with effective tools for managing finely grained concurrency is currently a very active research area, largely driven by the increased commercial availability of multi-core processors. However, there is, in our opinion, no clear consensus on what the right programming model and the right concurrency primitives are, and it is not surprising that the same situation holds for programming languages related to AgentSpeak.

Considering the eJason solution, for instance, it, in our opinion, represents a step forward in that it defines formally, in the eJason semantics, the behaviour of the new construct. On the other hand, to program highly concurrent agents by sharing a single critical section is likely to prove too inefficient in practise.

Borrowing inspiration from Java again, apart from the critical sections, whether labelled to permit more concurrency or not, we find the locks and conditions provided by the `java.util.concurrent.locks` library, which permits the programming of more flexible locking policies compared with basic critical sections. One interesting adaptation of that library is represented by the work on shared resources [5], where concurrent executions are guarded by concurrency preconditions, such that the execution of a resource blocks until its concurrency precondition (which are general predicates on the resource state) becomes true. As an item of future work, it would be interesting to implement this approach in eJason, essentially labelling critical section with general predicates over the agent state restricting access. Of course, in such an approach, care has to be taken in order to ensure efficient execution.

3 Executing Selected Event Plans in Matching Contexts

Consider a simplified multi-agent system with a classical client-server architecture. The client agents should write information into different files. In order to avoid conflicts generated by simultaneous write attempts to the same file by different agents, the access to the file is managed by a server agent. A client agent sends a message to the server agent to request the exclusive rights to access a file before it can write into such file. When the exclusive access to the file is no longer necessary, the client agent sends a message to the server agent to unlock the resource.

An implementation in Jason of the a client is depicted in Fig. 5. Before a client agent writes some text, `Text`, into a file, `FName`, it must first send an achieve message to the server requesting the lock over the file (i.e. delegating a goal of the shape `!lock(FName)` to the server). Then, it waits for the notification about the acquisition of exclusive access to the file. This notification is represented by a belief update event `+granted(FName)`. After the reception of this notification, the client agent writes the text into the file and requests the server to unlock the file (again, delegating this task as an achievement goal).

```
+!write(FName, Text) : true <-
   .send(server, achieve, lock(FName));
   .wait("+granted(FName)");
   write(Text, FName);
   .send(server, achieve, unlock(FName)).
```

Fig. 5. Jason code for the client agents

The Jason code for the server agent is shown in Fig. 6. The plan, referred to as `PSrv1`, handling the achievement goal to lock some file, `FName`, delegated from some client, `Client`, requires such a file to exist and not to be blocked by another agent. If these conditions hold, the first plan can be applied, which amounts to adding a mental note, `+blocked(Client, FName)`, recording that `Client` has exclusive access to the file `FName`. Then, it notifies the client by sending a tell message with the belief `granted(FName)`. The plan handling the achievement goal to unlock a file, referred to as `PSrv2`, checks whether the file exists and whether it is locked by the same agent that attempts to unlock it. The recipe provided by this plan implies erasing the aforementioned mental note that records the exclusive access granted to the agent `Client` over the file `FName` and, finally, notifying this client agent about the successful unlocking of the file.

Unfortunately, according to the semantics of Jason, a possible execution of the above server, in the presence of another unrelated intentation (I), is the following:

1. Two clients ($c1$ and $c2$) attempt to lock the same file, and send lock requests to the server.

```
+!lock(FName)[source(Client)] :    //PSrv1
   file(FName) & not blocked(_,FName)<-
       +blocked(Client,FName);
       .print("Agent ",Client," locks ",FName);
       .send(Client, tell, granted(FName)).

+!unlock(FName)[source(Client)] :  //PSrv2
   file(FName) & blocked(Client,FName) <-
       -blocked(Client,FName);
       .print("Agent ",Client," unlocks ",FName);
       .send(Client, tell, unlocked(FName)).
```

Fig. 6. Jason code for the file server agent

2. The server receives both lock requests, and selects the event corresponding to the request from ($c1$), selects the plan corresponding to the case where the server is not blocked, and instantiates an intention corresponding to that plan.
3. However, instead of executing the intention corresponding to the new event, the unrelated intention I is chosen instead.
4. Next, the event corresponding to the lock event by $c2$ is chosen, and the corresponding new intention is executed, thus locking the resource.
5. Finally, the intention corresponding to the lock request by $c1$ is executed, but *in a state where the plan context is no longer valid*, as the server is now blocked (by beginning serving request $c2$).

The problem here stems from the fact that the evaluation of a plan context and the execution of its plan body are decoupled. The Jason semantics allow several iterations to take place between the one in which the plan context is evaluated and deemed applicable; and the iteration in which the plan is chosen for execution.

In our opinion this is a severe problem, making it quite hard to write reliable event handling code.

3.1 Jason implementation solution: always select event intentions

In the current Jason implementation this problem is addressed in the default intention selection function, by always placing the intention updated in the reasoning cycle first in the "intention queue", and using a round-robin intention scheduling strategy [1]. Note that a programmer can still replace this default intention selection function with another one which, albeit faithful to the semantics, suffers from the above problem.

3.2 eJason solution: consecutive evaluation and execution of a plan

The original solution implemented in eJason was to examine the agent plan chosen to handle a particular event, and with a satisfied context. If this plan

begins with a critical section, the corresponding intention is always executed first. Otherwise the evaluation of the context and the execution of the intention is, potentially, decoupled.

3.3 A better solution: modifying the Jason semantics

In retrospect the eJason implementation is not particularly satisfying. Having to specify a critical section for potentially every agent plan can quickly produce quite ugly code. On the other hand the current Jason implementation solution is far from perfect too, as it, as far as we can understand always gives priority to event handling over executing other intentions. Moreover, in our opinion, it would be beneficial to modify the Jason semantics to remove the doubt whether a Jason implementation may ever decouple the execution of the plan context from beginning to execute the plan body. Permitting an interleaved execution of these basic plan steps just makes the Jason programmer's task unreasonably hard, with little gain.

Thus we argue for a Jason semantics change which: (i) strongly couples the execution of the plan context with the evaluating the first part of the plan body, and (ii) does not give priority to handling events compared to executing intentions. The resulting semantics will be published in a forthcoming publication.

4 Ensuring that Achievement Goals are not Dropped

In the example in Fig. 6, whenever a server agent gets the lock over a file, the requests from different client agents to lock the same file are disregarded, i.e., simply dropped, by the server agent, since the context of the relevant plan `PSrv1` cannot be satisfied.

This is another illustration of the difficulties posed by concurrent, or non-deterministic execution. That is, if we cannot precisely control the order and timing in which beliefs are asserted, or retracted, we may easily fail to predict situations (system states) where a goal may incorrectly be dropped because its context is not satisfiable. In other words, we risk creating *fragile* programs which normally work well but, in rarely encountered scenarios, fail. For programming such concurrent systems we believe it would be advantageous to have goal matching mechanisms that are less sensitive to the way the belief base changes over time.

As a second example, illustrating the difficulties in programming plans that are robust to all different situations where they may be tried, consider the agent below:

```
at(office). // Initial belief

!go(home). // Initial goals
!read(book).
```

```
+!read(Item): at(home) <-
  read(Item).

+!go(home): at(office) <-
  drive(home).
```

This agent is initially at the office and possesses, simultaneously, the desires of going home and reading a book. There are two possible outcomes for the execution of this agent. In both of them, the agent goes home (as the plan to accomplish such desire is always applicable in the initial state). However, the agent does not always satisfy its desire of reading a book, since the plan for doing so may be evaluated too soon (in the office), and thus dropped.

A programmer intending the agent to achieve both goals has to ensure that the goals are selected in the desired order. This requires considering all the possible interleavings and implementing some suitable synchronisation mechanism. The complexity of this synchronisation increases exponentially with respect to the number of goals to synchronise.

4.1 Jason solution: explicitely requeue achievement goals

The simplest solution to avoid dropping all the achievement goals that cannot be immediately handled due to the lack of applicable plans implies recording them within the agent's mental state in order to posteriorly pursue them. This solution can be achieved, e.g., adding a new plan, `PSrv3`, whose context matches whenever the file is already blocked by a different client agent. Following this plan, the server agent records the requests that cannot be immediately served by, e.g. returning the achievement goal addition event to the set of events:

```
+!lock(FName)[source(Client)] :    //PSrv3
   file(FName) &  blocked(_,FName)<-
   !lock(FName)[source(Client)]. // requeue
```

This solution, and similar ones, require the programmer to introduce a number of "fail-back" plans whose context is satisfied whenever the contexts of other (preferred) plans are not. This can obscure the code, and moreover, is dangerously fragile as it is easy to overlock situations where plans may fail due to nonsatisfied contexts.

Besides, note that by just enqueueing again the selected achievement goal addition event, the mental state of the agent is reinstated after a complete iteration of the reasoning cycle, hence consuming computational resources without changing the aforementioned mental state. Moreover, given the non-determinism of the selection functions, this event may be selected in consecutive iterations of the reasoning cycle, possibly enqueueing it in all such iterations (this behaviour is guaranteed if the belief base of the agent has not been updated, e.g. by other intentions, in-between), degrading the performance of the agent.

4.2 eJason solution: requeuing not applicable achievement goals

The alternative approach proposed in this article implies not generating a failure event for achievement goal addition events (i.e. events of the shape $\{+!g,\iota\}$). Instead, when one of these events is selected and there are no relevant or applicable plans for it, the event is returned to the agent's set of events (i.e. requeued). This way, the programmer does not need to be concerned about the timing of the selection of these events as they can be selected again later. For instance, the agent in the simple example above would always go home and read a book (i.e. avoiding possible race conditions), hence resulting in a single possible outcome for the agent program.

Note that it could be the case that the intention of the programmer were to provide a means for the agent to abandon its desire of reading a book when not at home. In our opinion, such behaviour shall not be relied upon the randomness of the intention selection function. Instead, the constructs already provided by the language, like the internal action *.drop_intention*, should be used. The following example shows how this construct can be used in this case:

```
at(office). // Initial belief

!go(home). // Initial goals
!read(book).

+!read(Item): at(home) <-
  read(Item).

+!read(Item): not at(home)<-
  .drop_intention(read(Item)).

+!go(home): at(office) <-
  drive(home).
```

Note that this alternative semantics is also available in the Jason implementation, by enabling a special configuration parameter, `requeue`, at startup. However, our proposal is to declare this alternative semantics *the standard Jason semantics*, as is the case in eJason.

5 Suspending Test Goals

Similar to the situation with achievement goals, whenever a programmer introduces a test goal, $?g$, into the body of a plan, p, the programmer must consider the possibility that such a test goal may fail (along with the whole plan). This failure occurs if the test, g, cannot be satisfied when the corresponding test goal addition event, $\{+?g,\iota\}$, is selected by the agent's event selection function. Many Jason programs, we believe, could be written more clearly if test goals that cannot be satisfied are "suspended" until they become valid.

5.1 eJason solution: providing a new suspensful test operator

In eJason we have introduced a new operator "??" for expressing that we want a test goal to suspend until it is satisfiable. The semantics of a goal $??g$ is similar to the semantics of a $?g$ test goal. The difference lies in the treatment that the corresponding goal addition event, respectively $\{+??g,\iota\}$ and $\{+?g,\iota\}$, receives from the eJason interpreter. When the test g cannot be satisfied using the information in the agent's belief base, for some test goal $\{+??g,\iota\}$, this event is returned to the agent's set of events (note the similarity to the proposed semantics for achievement goal addition events). Therefore, this event will be selected again at a later iteration of the reasoning cycle.

To illustrate the behaviour of the operator let us code an agent that delegates some tasks $t1$ and $t2$ to other agents *alice* and *bob*, respectively, and then gathers the result of executing the tasks:

```
gather_results(Res1,Res2) :-
  result(t1, Res1) & result(t2, Res2).

+!task3(Result) <-
  .send(alice, achieve, t1);
  .send(bob, achieve, t2);
  ??gather_results(Res1,Res2);
  operation(Res1,Res2,Result).
```

Note that the new operator "??" is used to introduce a mechanism to suspend the execution of an intention until some conditions are met. This mechanism provides a functionality similar to that of the internal action `.wait(Event)`. However, while `.wait` relies on the occurrence of a single event or a logical expression (e.g., querying the belief base), as condition for the reactivation of an intention, the operator "??" establishes a goal g that must be matched in order to reactivate the intention.

The new operator can help simplify code, as shown in Fig. 7, where the behaviour of the client agent introduced in Fig 5 no longer requires the inclusion of two separate, though semantically dependent, plans.

```
+!write(FName, Text) :true <-
   .send(server, achieve, lock(FName));
  ??granted(FName);
   write(Text, FName);
   .send(server, achieve, unlock(FName)).
```

Fig. 7. Modified client agents, using the "??" operator

6 The Reasoning Cycle of eJason agents

The introduction of the new language constructs and semantics described above alters the agent's interpreter, i.e., the number of state transitions in the Jason reasoning cycle have increased. In Fig. 8 the new transitions are depicted as dashed lines. The different steps that compose the reasoning cycle of a Jason agent are the following:

- **ProcMsg:** during this initial step, the agent obtains information from its environment (perception) and from the messages received from other agents. This information may update the belief base and provide new goals, generating the corresponding events in each case.
- **SelEv:** one of the unprocessed events is chosen to be processed during the current iteration. Such event is selected using an agent-specific *event selection function*.
- **RelPl:** the set of relevant plans for the event selected in the previous step is computed. If there is no relevant plan either the event is discarded (in the case of belief additions/deletions) or a failure event is generated (in the case of goal additions).
- **ApplPl:** the set of applicable plans is computed from the set of relevant plans. If the set of applicable plans is empty, either the selected event is discarded or a failure event is generated (for the same cases as before).
- **SelAppl:** one, and only one, of the applicable plans is selected using the agent-specific *option selection function*.
- **AddIM:** if the selected event possesses a related intention (i.e. it is a subgoal added during the execution of an instruction in the body of another plan), the selected applicable plan is put on top of such intention (recall that an intention is a stack of plans). Otherwise, a new intention, only containing the selected applicable plan, is added to the set of intentions.
- **SelInt:** if the set of intentions is empty, then a transition to the inital step is taken. Otherwise, an agent-specific *intention selection function* selects one intention from the set of intentions of the agent. Note that each intention represents a different focus of attention of the agent.
- **ExecInt:** the first formula in the body of the plan on top of the intention stack is executed, triggering some modification of the agent's mental state or its environment (e.g. adding a new belief or sending a message), along with the generation of the corresponding event. If the formula executed is not a goal addition of type `!g` or `?g`, such formula is removed from the body of the plan. Otherwise, the execution of the intention is suspended until the subgoal introduced is fully executed (the suspended intention appears as a related intention to the corresponding goal addition event). Note that only one formula is executed in each iteration of the cycle.
- **ClearInt:** if the body of the plan on top of the intention is not empty (i.e. the plan has not been fully executed) the intention is returned to the set of intentions and a new iteration starts. Otherwise, such plan is removed from the top of the intention stack. If there are more plans left in the intention, a

transition to the inital step is taken. If the intention is empty, it is completely removed and a new iteration starts.

The new transitions, depicted in Fig. 8 as dashed transitions, are the following:

- When there are no relevant plans for either an achievement goal or a test goal introduced using the operator "??", the transition k is taken.
- During the execution of a critical section no new events are selected. After executing an action within a critical section, the transition l is taken.
- During the execution of a critical section, the focus of attention does not change. After the addition of the intended means for a goal within a critical section, the transition m is taken.
- During the execution of critical sections, the presence of failures introduces new transitions. The absence of relevant plans for an event causes transition n. The emptiness of the set of applicable plans causes transition o.

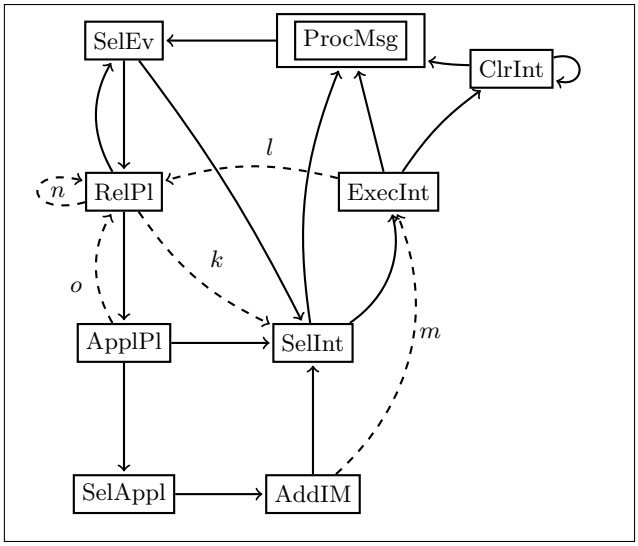

Fig. 8. Possible state transitions within a reasoning cycle in eJason.

7 Conclusions

In this paper we have analysed a number of Jason mechanisms for controlling intra-agent concurrency and communication, and have identified a number of potential pitfalls these mechanisms can cause an inexpert Jason programmer. Moreover, we have suggested alternatives to these mechanism, which, in our

opinion, may make the task of controlling and coordinating the concurrent activities of Jason (intra-agent) intentions easier.

For two of these mechanisms we advocate changing the standard Jason sematics. An alternative to doing so is to *configure* a standard Jason implemententation by replacing e.g. the standard intention selection function with a custom one. However, we argue that there are dangers in such customizations too, as a (concurrent) Jason program cannot then be judged correct by itself, but must be judged in conjunction with the particular configuration it is designed to be run under.

In future work we aim to revise the Jason semantics to account for these new mechanisms, without relying on external customization functions.

Acknowledgments

This work has been partially funded by the Spanish MINECO project TIN2012-39391-C04-02 *STRONGSOFT*, and the Madrid Regional Government grant S2013/ICE-2731 *N-GREENS*. Authors would also like to thank Rafael H. Bordini for his useful comments.

References

1. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons (2007)
2. Collier, R.W., Russell, S., Lillis, D.: Reflecting on agent programming with agentspeak(l). In: Chen, Q., Torroni, P., Villata, S., Hsu, J., Omicini, A. (eds.) PRIMA 2015: Principles and Practice of Multi-Agent Systems. pp. 351–366. Springer International Publishing, Cham (2015)
3. Fernández-Díaz, Á., Benac Earle, C., Fredlund, L.A.: eJason: an implementation of Jason in Erlang. pp. 7–22. Proceedings of the 10th International Workshop on Programming Multi-Agent Systems (ProMAS 2012) (2012)
4. Fernandez Díaz, A.: EJason: A Framework for Distributed and Fault-tolerant Multi-agent Systems. Ph.D. thesis, Universidad Politécnica de Madrid (2018)
5. Fredlund, L.Å., Mariño, J., Alborodo, R.N., Ángel Herranz: A testing-based approach to ensure the safety of shared resource concurrent systems. Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability **230**(5), 457–472 (2016). https://doi.org/10.1177/1748006X15614231, https://doi.org/10.1177/1748006X15614231
6. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) Agents Breaking Away, LNCS, vol. 1038, pp. 42–55. Springer (1996). https://doi.org/10.1007/BFb0031845, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 Jan. 1996, Proceedings
7. van Riemsdijk, M.B., Dastani, M., Meyer, J.C.: Goals in conflict: semantic foundations of goals in agent programming. Autonomous Agents and Multi-Agent Systems **18**(3), 471–500 (2009). https://doi.org/10.1007/s10458-008-9067-4, https://doi.org/10.1007/s10458-008-9067-4

8. Shapiro, S., Sardiña, S., Thangarajah, J., Cavedon, L., Padgham, L.: Revising conflicting intention sets in BDI agents. In: van der Hoek, W., Padgham, L., Conitzer, V., Winikoff, M. (eds.) International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes). pp. 1081–1088. IFAAMAS (2012), http://dl.acm.org/citation.cfm?id=2343851
9. Thangarajah, J., Padgham, L.: Computationally effective reasoning about goal interactions. J. Autom. Reasoning **47**(1), 17–56 (2011). https://doi.org/10.1007/s10817-010-9175-0, https://doi.org/10.1007/s10817-010-9175-0
10. Zatelli, M.R., Hübner, J.F., Ricci, A., Bordini, R.H.: Conflicting goals in agent-oriented programming. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 21–30. AGERE 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/3001886.3001889, http://doi.acm.org/10.1145/3001886.3001889