AgentSpeak(ER): An Extension of AgentSpeak(L) improving Encapsulation and Reasoning about Goals

Paper #XXX

ABSTRACT

In this paper, we introduce AgentSpeak(ER), an extension of the AgentSpeak(L) language tailored to support encapsulation. The AgentSpeak(ER) extention allows for significantly improving the style of BDI agent programming along relevant aspects, including program modularity and readability, failure handling, and reactive as well as goal-based reasoning. The paper introduces the novel language, formalises the changes in the usual semantics of AgentSpeak, illustrate the features of the language through examples, and shows results of experiments evaluating the proposed language.

CCS CONCEPTS

• **Computing methodologies** → **Intelligent agents**;

1 INTRODUCTION

AgentSpeak(L) has been introduced in [**?**] with the purpose of defining an expressive, abstract language capturing the main aspects of the Belief-Desire-Intention architecture [**?** **?**], featuring a formally defined semantics and an abstract interpreter. The starting point to define the language were real-world implemented systems, namely the Procedural Reasoning System (PRS) [**?**] and the Distributed Multi-Agent Reasoning System (dMARS).

AgentSpeak(L) and PRS have become a main reference for implementing concrete Agent Programming Languages based on the BDI model: main examples are Jason [**?** **?**] and ASTRA [**?**]. Besides Agent Programming Languages, the AgentSpeak(L) model has been adopted as the main reference to development several BDI agent-based frameworks and technologies [**?** **?**] as well as serving as inspiration for theoretical work aiming to formalise aspects of BDI agents and agent programming languages [**?** **?** **?**].

Existing Agent Programming Languages extended the language with constructs and mechanisms making it practical from a programming point of view [**?**]. Besides, proposals in literature extended the model is order to make it effective for specific kinds of systems — e.g., real-time systems [**?**], XXX — or to improve the structure of programs, e.g. in terms of modularity [**?** **?**].

Along this line, in this paper we describe a novel extension of the AgentSpeak(L) model — called AgentSpeak(ER) — featuring *plan encapsulation*, i.e. the possibility to define plans that fully encapsulate the strategy to achieve the corresponding goals, integrating both the pro-active and the reactive behaviour. This extension turns out to bring a number of important benefits to agent programming based on the BDI model, namely:

- encapsulation of goal-oriented behaviour – ...
- modularity of the agent code – ...
- failure handling – ...
- runtime management of goals and intentions – ...

Besides the benefits in terms of agent programming, the approach:

- reduces the gap between the design level and the programming level, ...
- facilitate goal-based reasoning – ...

The remainder of the paper is organised as follows: first we describe in details the motivations that lead to the proposal (Section 2); then, we introduce and discuss AgentSpeak(ER), introducing the main concepts, syntax and semantics — first informally (Section 3) and then providing the formalisation of some key aspect (Section 4). Finally we discuss the results of a first evaluation that has been carried out, based on a prototype implementation extending the ASTRA platform (Section 5).

2 MOTIVATION

The main motivation behind AgentSpeak(ER) comes from the experience using agent programming languages based on the AgentSpeak(L) model, Jason and ASTRA in particular. Yet, these issues are relevant for any language based on the BDI architecture. The first issue is about the plan model.

In the BDI model, plans are meant to specify the means by which an agent should satisfy an end [**?**]. In AgentSpeak(L) a plan consists of a rule of the kind `e : c ¡- b`. The head of a plan consists of a triggering event `e` and a context `c`. The triggering event specifies why the plan was triggered, i.e., the addition or deletion of a belief or goal. The context specifies those beliefs that should hold in the agent's set of base beliefs, when the plan is triggered. The body of a plan is a sequence of actions or (sub-)goals.

So in this approach — as well as in planning, in general — the *means* to achieve a goal — i.e., the body — is meant to be fully specified in terms of the actions the agent should execute and the (sub-)goals the agent should achieve or test. In the programming practice, however, it is often the case that the strategy (the means) adopted to achieve some goal (the end) would naturally include also *reactions*, i.e., reacting to events asynchronously perceived from the environment, including changes about the beliefs. That is, an agent is not reacting only to unexpected events so as to change/adapt

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. Andre, S. Koenig (eds.), July 2018, Stockholm, Sweden
© 2018 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.
https://doi.org/doi

its course of actions, but reactivity could be an effective ingredient of the strategy, of the plan adopted to achieve some goal. Besides the programming practice, this is often the case if we consider human activities too. Reactivity is a key ingredient of many activities that we perform to achieve specific goals, not only to handle events that represents errors or unexpected situations (for the current courses of actions). It follows naturally that this is also an opportunity to extend the plan model so as to fully *encapsulate* also reactions that are part of the strategy to achieve the goal, as well as the subgoals that are specific to that particular goal.

In the following, let's refer to plans triggered by event goals `+!g <- ...` as *g-plans*, and plans triggered by the belief change (including percepts) as *e-plans*. The BDI reasoning cycle automatically updates beliefs from percepts, and this allows us to write down structured plans with courses of actions that change according to the environment, by exploiting goals/subgoals and contexts. For instance, as a very simple example, yet capturing the point, let us consider the goal of printing down all the numbers between N and 1, stopping if/when a 'stop' percept is perceived. This task can be effectively tackled using only g-plans:

```
+!print_nums(0).
+!print_nums(_) : stop.
+!print_nums(N) : not stop <- println(N); !print_nums(N-1).
```

This is a clean solution, exploiting the context in plans and the fact that in the reasoning cycle beliefs are automatically updated given new percepts. More generally, in the AgentSpeak(L) model the suggested approach to realise structured activities whose flow can be environment dependent is by means of (sub-)goals and corresponding plans with a proper context.

In some cases however this approach is not fully effective, and *e-plans* are needed, in particular every time we need to react *while executing the body of a plan*. A typical case occurs when we have actions (or subgoals) in a plan that could take a long time to complete. For instance, suppose that instead of simply printing we have a long-term `elab` goal (it could be even an action):

```
+!elab_nums(0).
+!elab_nums(N) : stop.
+!elab_nums(N) : not stop <- !elab(N); !elab_nums(N-1).
```

Suppose that, realistically, we cannot spread/pollute plans about the goal `!elab` with a stop-dependent behaviour. To solve this problem, using the AgentSpeak(L) model and in BDI architectures an e-plan can be introduced, using e.g. internal actions to act on the current ongoing intention:

```
+!elab_nums(0).
+!elab_nums(N) <- !elab(N); !print_nums(N-1).
+stop <- .abort(elab_nums(_)).
```

The problem here is that the plan `+stop <- ...` is part of the strategy to achieve the goal `!elab_nums`, however it is encoded as a separate unrelated plan.

The use of e-plans to achieve goals is actually an important conceptual brick of the AgentSpeak(L) model. Let us consider the robot cleaning example used to describe plans in [?]. One of the plans is:

```
+location(waste,X):location(robot,X) & location(bin,Y)
  <- pick(waste); !location(robot,Y); drop(waste).
```

That is, as soon as the robot perceives that there is a waste in its location, then it can pick it up and bring it to the bin. This plan an essential brick of the overall strategy to achieve the goal of cleaning the environment — or to maintain the environment clean, in a *maintenance goal* view. However, the fact that this plan is useful to achieve that goal is not explicit in the source code, it remains in the mind of the programmer/designer. The agent reacts to a waste in a location because it has an implicit goal, which is about cleaning the environment. However, since there are no g-plans about it, there is no explicit trace in the agent mental structures about this goal.

This issue impacts also on plan failures handling, which is a very important aspect of agent programming. In the Jason dialect of AgentSpeak(L), we can define plans that handle the failure of the execution of g-plans (generating the event `-!g`), but not of e-plans. So, for example, if there is a problem in the println action in:

```
+stop <- println("stopped").
```

the plan execution fails, without any possibility to react and handle the failure. In order to handle this, a programmer is forced to structure every e-plan with failure handling using a subgoal:

```
+stop <- !manage_stop.
+!manage_stop <- println("stopped").
-!manage_stop <- println("failure").
```

This contributes to making the program longer and verbose, besides increasing the number of plans to be managed by the interpreter.

To summarise, this is the set of key issues identified for the basic plan model in the practice of agent programming:

- **Lack of encapsulation:** The strategy to achieve a goal is fragmented among multiple plans, not explicitly related.
- **Implicit vs. Explicit goals:** A part of the plans in a program have a goal which is not explicit, it is implicit.
- **Difficult failure handling:** Failures/errors generated in the body of e-plan cannot be directly captured.

This has an impact also on the performance of the agent interpreter at runtime. In the traditional model, each reaction represented by an e-plan necessarily generates a new intention to manage it, increasing the number of plans in execution to be managed by the agent reasoning cycle. This could have also a negative impact on intention selection at each reasoning cycle [?].

Besides the pure programming level, it is worth noting that these issues can affect the software engineering process concerning agent development. In particular, at the design level, it is natural to specify coarse-grained plans fully encapsulating the strategy to achieve or maintain goals. It would be important then to keep as much as possible the same level of abstraction when going from the design to programming, and at runtime too, to support agent reasoning.

3 THE AGENTSPEAK(ER) PROPOSAL

To overcome the problems discussed above, we consider two key changes in the plan model.

The first one is to extend the body of plans beyond the simple sequence of actions and goals, so as to include also the possibility to specify a reactive behaviour encapsulated in the plan. Coherently with the AgentSpeak(L) model, such a behaviour can be expressed in terms of e-plans. Accordingly, a plan body becomes the *scope* of both a sequence of actions – referred as *main sequence of actions* – and a set of e-plans, specifying a reactive behaviour which is active at runtime only when the plan is in execution, and g-plans, specifying plans to achieve goals that are relevant only in the scope of this plan. The e-plans set may include also reactions to failures occurring when executing actions of the main sequence or the body of e-plans.

The second one is enforcing e-plans – as pieces of reactive behaviours – must always be defined in the context of a g-plan – that is we enforce the principle that an agent does (and reacts) always because of a goal to achieve or maintain. This enforce the programmers to explicitly specify what's the goal to achieve even when defining a pure reactive behaviour. In so doing, at runtime every intention – i.e., plan in execution – is associated to a goal to achieve.

In the remainder of the section we first describe in detail the syntax and informal semantics of the new plan model, including simple examples, and then we discuss the key benefits.

3.1 Informal Syntax and Semantics

At the top level, an AgentSpeak(ER) program is a collection of g-plans, whose syntax is shown in the following:

```
/*  g-plans */
+!g : c <: gc {

  <- a; b; ?g1; !g1; !!g2. // main sequence, optional.

  /* e-plans */
  +e1 : c1 <- b1.
  +e2 : c2 <- b2.
  +!k : true <: b(10) {
    <- a,b,c.
    +e3 : c3 <- b3. // possible old-style plans
  }

  /* e-plans catching failures */
  -!g[error(ia_failed)] : ...
    <- ... catches from failures

  /* further g-plans */
  +!g1 : c1 <: gc1 { ... }
}
```

Like in AgentSpeak(L), a g-plan is defined by rule with a head and a body. In AgentSpeak(ER) the body is defined by two elements: a *plan block*, including a main sequence and a set of plans, and a *goal condition* gc—which is a belief formula analogous to the context. The main sequence of actions can be empty—this is the case of plans expressing a pure reactive behaviour.

Conceptually, the sub-plans (e-plans and g-plans) stored in the plan block are available in the plan library only when the g-plan is an execution. Like in AgentSpeak(L), as soon as the g-plan is instantiated, the main sequence is executed. If no goal condition is specified, the plan is considered completed as soon as the main sequence is completed. Otherwise, the g-plan is considered completed only if/when the beliefs expressed in the goal condition hold.

While a g-plan is in execution, if the agent perceives an event triggering a sub-plan described in the plan block and the sub-plan is applicable according to its specified context, then the body of the plan is stacked on top of the stack of the current intention (associated to the g-plan). If the agent was executing the main sequence, the effect is to *interrupt* it so as to execute the body of the sub-plan. In AgentSpeak(L) this is the case when executing a sub-plan associated to a goal defined in the sequence, e.g.: a1; a2; !g1; a3, in a synchronous way. In AgentSpeak(ER) this uniformly occurs also with any kind of events, not only goal events, that is: the main sequence can be *asynchronously* interrupted to react to events coming from the environment. This behaviour is ruled by the (extended) reasoning cycle, which will be described in next section.

[TO BE COMPLETED]

3.2 Examples

Let's consider the examples seen in 2, to see how they be encoded in AgentSpeak(ER). Robot cleaning example becomes:

```
+!clean_env <: false {
  +location(waste,X):location(robot,X) & location(bin,Y)
    <- pick(waste); !location(robot,Y); drop(waste).
  ...
}
```

That is: now we can give an explicit reason to the reactive behaviour by encapsulating the e-plan inside a g-plan, with an explicit goal. Setting the goal condition to false means that the plan execution is going to last for ever. It is a way to implement kind of maintenance goal. As a further note, this is a particular case in that the plan body has no main sequence.

The Elab num example:

```
+!elab_nums(0).
+!elab_nums(N) <: stop || .main_done {
  <- !elab(N); !elab_nums(N-1).
  +!elab(M) <- ...
}
```

.main_done is a predefined meta belief, which holds when the main sequence has been executed.

In the case that we want to do some actions when reacting to stop, then we can introduce a e-plan as follows:

```
+!elab_nums(N) <- {
  <- !elab(N);
```

```
      !elab_nums(N-1).
  +!elab(N') <- ...
  +stop
    <- println("stopped"); .done.
}
+!elab_nums(0).
```

The approach allows to avoid towers of subgoal calls, that are used when a plan do some action and then should wait that some condition is achieved, in order to complete, eventually reacting to some events to adapt the course of actions. An example is the simple thermostat. Let's consider the goal to achieve some temperature +!temp(T):

```
+!temp(T) : temp(CT) & CT < T & not warming
  <-  switchOnWarming; !temp(T).
+!temp(T) : temp(CT) & CT > T & not cooling
  <-  switchOnCooling; !temp(T).
+!temp(T) : temp(CT) & CT != T
  <-  !temp(T).
+temp(T) : temp(T) <-  stop.
```

There is here a kind *cognitive busy waiting* situation: after switching on the HVAC system (to cool or to warm), then the strategy needs to continuously check the condition to know when/if the goal has been achieved, before ending the plan. This is problematic for 2 reasons: the size of the intention stack is growing indefinitely, and performance overhead.

A solution with the extended plan model:

```
+!temp(T) : temp(CT) & CT < T
  <: temp(T) & not warming & not cooling {
  <- switchOnWarming.
  +temp(T) <- stop. // There is bound
}

+!temp(T) : temp(CT) & CT > T
  <: temp(T) & not warming & not cooling {
  <- switchOnCooling.
  +temp(T) <- stop.
}
+!temp(T) : temp(T).
```

Remarks:

- the T variable used inside the plan body (+temp(T) ¡-..) has been bound in +!temp(T)

In Jason, the original AgentSpeak(L) solution could be avoided by exploiting a .wait internal action that suspends a plan until some condition is achieved:

```
+!temp(T) : temp(CT) & CT < T
  <-  switchOnWarming;
      .wait(temp(T));
      stop.

+!temp(T) : temp(CT) & CT > T
  <-  switchOnCooling;
      .wait(temp(T));
      stop.
```

However this solution is no more effective (alone) as soon as something could happen while waiting that need to change the course of actions. For instance, suppose that while warming the temperature value surpasses the target value T, such that we would need to start cooling. In this case, the .wait solution must be strongly reworked and becomes complicated. The complete solution using the extended plan model:

```
+!temp(T) <: temp(T) & not warming & not cooling {
  +temp(T) <- stop.
  +temp(T1) : T1 > T <- switchOnCooling.
  +temp(T1) : T1 < T <- switchOnWarming.
}
```

Maintenance goal: to achieve and keep the temperature at the target temperature T, with the possibility of dynamically updating/changing T.

```
+!target_temp(T) <: false {
  <- +target_temp(T).

  +temp(T) : target_temp (T) <- stop.
  +temp(T1) : target_temp(TT) & T1 > TT <- switchOnCooling.
  +temp(T1) : target_temp(TT) & T1 < TT <- switchOnWarming.

  +target_temp(TT) : temp(TT)  <- stop.
  +target_temp(TT) :
    temp(T) & T < TT & not warming <- switchOnWarming.
  +target_temp(TT) :
    temp(T) & T > TT & not cooling <- switchOnCooling.
}
```

Failures generated by the execution of actions of sub-plans failures can be handled and managed:

- by sub-plans listed in the body of the plan
- by plans at the same level of +!g, if the event is not managed inside the body

In the latter case, the event representing the failure could be exactly -!g as in Jason, i.e. the event which is generated when the goal is removed. This strategy cannot be adopted in the former case, since we want to react to a failure but the corresponding goal and intention have not been removed yet, so no -!g event is generated. For this reason, we need to introduce a new event: `+/err_term` [TEMPORARY SYNTAX] representing the failure of an action belonging to the main sequence or to the body of subplans.

```
+!g : c <: GoalCondition {
  <- a; b; ? ; c.
  ...
  +/err_term : cn <- bn
}
```

In the cycle, the `+/err_term` is treated as the other events: if an applicable plan is found, it is selected and executed (in the same intention). Otherwise, the behaviour is like it is in pure AgentSpeak(L) - the whole intention is removed and a `-!g` event is generated.

3.3 Discussion

[TO BE COMPLETED]

In this section we go back on the key issues, showing how the approach is effective in overcoming them. Besides, we explicit discuss further goodies that the model brings in agent programming.

3.3.1 *Full encapsulation.*

3.3.2 *Explicit vs. Implicit Goals.*

3.3.3 *Failure Handling.*

3.3.4 Coarse-grained Intentions. In AgentSpeak, each e-plan in execution has its own intention/stack, which runs concurrently to the other intentions. Conceptually, this follows the idea that that the management of environment events are not part of an existing plan to achieve some goal. In the new model instead, an e-plan inside a g-plan is meant to specify a behaviour useful for achieving the goal of the g-plan, so part of the same intention. For this reason, the an e-plan (subplan) inside a g-plan in execution is triggered, no new intentions are generated and the body of the sub-plan is placed on the top of the stack of the same intention. The new model leads than to have more coarse-grained intentions, reducing overall the number of intentions. This has an impact on the reasoning cycle, for instance on the step selecting the intention to carry on.

3.3.5 Intention implicit Interruption. The effect is to implicitly interrupt and suspend the main sequence (body) of the g-plan - if available and if not already completed. Conceptually, this is the asynchronous version of the suspension that occurs when a subgoal is specified (!g) in the body of a plan. This causes exactly the suspension of the body and the pushing of the body of the new plan on the top of the stack of the intention.

3.3.6 *Compatibility with AgentSpeak(L).*

4 SEMANTICS

Briefly, these are the changes in the semantics of AgentSpeak(L) that are needed to accomodate the new features of AgentSpeaker(ER):

- in the beginning of the reasoning cycle, check all goal-conditions starting from the bottom of each intention stack and remove all finished intentions;
- if an event is *external*, try to trigger one relevant/applicable plan for each intention stack, starting from the top of the intention stack (that is, the most specific plan in the g-plan tree);
- if an event is *internal*, search for a relevant applicable plan on the path to the root of the g-plan tree, that is, starting from the most specific plans in the g-plan tree (as in the item above but now there is only one intention to refer to); this can be done using prefixes to goal names and does not change the semantics per se, only the order in which we check for relevant applicable plans.

We start by updating the well-known reasoning cycle [**?**] to adapt it for the initial stage of checking all goal-conditions to remove the intentions that should no longer be active. Recall that this can be because the goal has been achieved, has become impossible to achieve, or the motivation why it was adopted no longer applies. Furthermore, note that even though this is computationally costly, it is necessary because not doing it at every reasoning cycle may imply the agent missing the moment where the goal-condition became true (hence the g-plan needing to be deactivated); it is also worth the computational burden in as much as it has the various practical programming advantages we pointed out earlier in this paper.

The required stage for checking goal-conditions is included in the existing ClrInt stage (which previously only removed empty intentions) except it is now moved to the beginning of the reasoning cycle (to ensure nothing in the reasoning cycle is done under the assumption a deactivated goal is still being pursued), just after the ProcMsg stage (as the information just received from other agents might be useful in checking for goals to be deactivated). The clearing of intentions used to be the last part of the reasoning cycle, but because there are no other dependencies between the first and last stages, ClrInt might as well be done at the beginning rather than the end. The slightly changed reasoning cycle is shown in Figure 1.

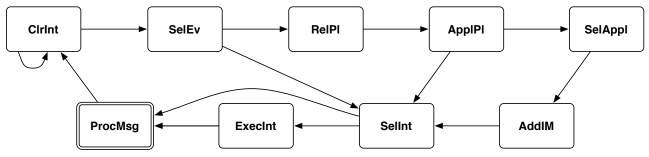

Figure 1: The AgentSpeak(ER) Reasoning Cycle

The ClrIntstage also needs a new semantic rule. To facilitate the presentation of that rule, we define a new auxiliary function GCond, which given a g-plan simply returns the goal-condition component of that plan, as follows. Let $p =$ "$+!\mathsf{g} : \mathsf{b} <: \mathsf{c} < -\{\mathsf{a}.\}$", then $\mathsf{GCond}(p) = c$, more specifically the logical formula coded by c. Also, we denote by $[p_1, ..., p_n]$ an intention stack with plan p_n at its top.

The new rule ClrInt_4, as shown below, is added to the 3 existing ones (see [**?** , p 212]). It essentially removes from intentions the bottom most g-plan for which ist goal-condition now follows from the state of the belief base. The intuition is that if goal `!g1` required a subgoal `!g2` (a plan for which was pushed on top of the plan for the former one in the intention stack) and `!g1` is no longer active, that whole part of the intention stack above it needs to be removed together with it.

$$\frac{i \in C_I \qquad i = [p_1, \ldots, p_n] \qquad ag_{bs} \models \mathsf{GCond}(p_j) \text{ for some } j, 1 \le j \le n}{\langle ag, C, M, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ClrInt}\rangle}$$

$$\begin{array}{lrcl} \textit{where:} & j & \text{is the least number in } [1..n] \text{ s.t.} & \\ & & ag_{bs} \models \mathsf{GCond}(p_j) & \\ & C'_I & = & C_I \setminus \{i\} \cup \{[p_1, \ldots, p_{j-1}]\} \end{array} \quad (\text{ClrInt}_4)$$

The auxiliary functions $\mathsf{RelPlans}(ps, te)$ $\mathsf{AppPlans}(ps, te)$ (see Definitions 10.2 and 10.3 in [**?**]) can be changed to accommodate both the new g-plan structure as well as the firing of reaction plans (i.e., plans for external events) for all intentions rather than creating a single new separate intention as before. First, the redefined functions now receive a set of intentions as an extra parameter; they are always called with the current contents of the set of intentions (C_I in the operational semantics). If te is an external event, the new parameter is used to check for relevant/applicable plans for each individual intention plus the empty intention $\top$. The empty intention being included to the set of intentions in the parameter ensures that relevant reaction plans for the "main" goal[1] will be checked; this in turn allows for backwards compatibility with AgentSpeak(L). Besides having an extra parameter, the functions now return a set of triples rather than a pair. We use $\mathsf{TrEv}(p)$ to refer to the triggering event of a plan p. Formally, we have:

Definition 4.1 (Relevant Plans). The auxiliary function to retrieve relevant plans given a plan library ps, a particular event te of type `+l` or `-l` (i.e., an external event, one reacting to changes in beliefs rather than a goal adoption event), and a set of intentions $\mathcal{I}$ is defined as $\mathsf{RelPlans}(ps, te, \mathcal{I}) = \{\langle ps, \theta, i\rangle \mid i \in \mathcal{I}, i = [p_1, \ldots, p_n]$, and j is the greatest number in $[1..n]$ s.t. p_j is a g-plan with a (reaction) plan rp in scope and $\{te\} \models \mathsf{TrEv}(rp)\theta\}$, for some m.g.u. θ. By "in scope" we mean that plan rp appears within the g-plan for p_j (as for example ??? in Figure **??**). For internal events the function returns a set containing a single element $\langle ps, \theta, i\rangle$ where i is the specific intention that generated te and ps is the set of relevant plans using the same idea of scope as above for external events.

We do not formally define the AppPlans function here due to space, and given that it is trivially extended in a very similar way as in Definition **??** for RelPlans.

Finally, we need to change the semantics so that not just one but *all* relevant reaction plans are triggered, that is, one for each active intention, and within a single intention starting from the most specific plan (i.e., the one closest to the top of the intention if there are relevant plans at other levels too). Most of the work was already done in the redefinitions of the RelPlans and AppPlans functions, but we still need to change the rule for processing external events (which now change multiple intentions). Note however that by the time we come to handle an external event with rule ExtEv, a single plan for each intention has already been selected; that is, $S_{\mathcal{O}}$ (the option selection function) is also slightly redefined to work with the new structures returned by those redefined auxiliary functions.

The new ExtEv rule is below, and it essentially says that an external event now potentially interrupts various intentions, provided there are relevant and applicable plans anywhere in the g-plans associated with each particular current intention of the agent. Recall that external events for the "main" goal — the external events as in traditional AgentSpeak — will now be associated with the empty intention $\top$in T_ρ(i.e., the result of the RelPlans and AppPlans functions). The chosen plan, after being applied its respective variable substitution θ, for each intention is pushed on top of that intention (the notation used for this below is $i_j[p_j\theta_j]$).

[1]Recall that this is a default g-plan that is implicitly associated with top-level plans, as in traditional AgentSpeak(L).

$$\frac{T_\varepsilon = \langle te, \top\rangle \qquad T_\rho = \{\langle p_1, \theta_1, i_1\rangle, \ldots, \langle p_n, \theta_n, i_n\rangle\}}{\langle ag, C, M, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ClrInt}\rangle}$$

$$\textit{where:} \qquad C'_I \;=\; C_I \setminus \bigcup_{j=1}^{n}\{i_j\} \cup \bigcup_{j=1}^{n}\{i_j[p_j\theta_j]\} \qquad (\text{ExtEv})$$

To conclude this section, we emphasise that, for the sake of space, we here formalised only the main changes to the well-known AgentSpeak semantics. The complete new transition system giving semantics to AgentSpeak(ER) should appear in a much longer paper.

5 EVALUATION

The new language has been evaluated through a prototype implementation that builds on the ASTRA [**?**] language. This prototype, which is compatible with pure ASTRA code, is used to reimplement part of an existing solution for the Minority Game [**?**]. Interpreter cycle timings are then captured to develop an initial understanding of how the new language performs.

As can be seen in the snippet of code below,the implementation of the AgentSpeak(ER) prototype is syntactically different to the example provided in section 3. This reflects the syntactic approach adopted in ASTRA. It does not impact the associated semantics.

```
g-rule +!g() : c <: gc {
  body {
    // main plan body goes here...
    a; b; !g1(); !!g2();
  }

  /* e-plans */
  rule +e1 : c1 { b1; }
  rule +e2 : c2 { b2; }
}
```

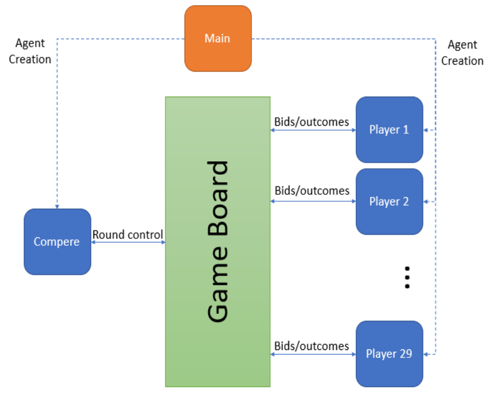

Figure 2: Minority Game Agent Architecture

5.1 Minority Game

The Minority Game is a well known model for evaluating collective behaviour of agents competing for finite resources [**?**]. In a nutshell, the game involves an odd number of agents competing for a resource over a series of rounds. For a round, each agent makes a binary decision (yes/no). At the end of the round, the bids are counted, and the agents that are in the minority win. The game has been applied mainly in the area of modelling financial markets [?].

To provide an initial evaluation of AgentSpeak(ER), we adapt an existing ASTRA-based implementation of the Minority Game MG). As is shown in figure 2, the implementation consists of 3 types of agent: the *compere* agent, who is responsible for managing the game (starting rounds, evaluating bids, ...); the *player* agent, who implements a set of MG strategies; and the *main* agent, which is responsible for configuring the game (creating and configuring the compere and player agents).

The existing implementation currently consists of 10 plans. Two of the plans are generic and are used for configuration. Another plan implements the bidding behaviour, delegating to a `!select_bid(...)` sub goal that can be contextually selected based on the current strategy. Of the remaining plans: four relate to the best play strategy; one relates to a random strategy; and two relate to a sticky strategy.

In the AgentSpeak(ER) implementation, we reduce this to 4 of the new g-rules: one to handle configuration and one for each of the strategies. The code below illustrates this new model through the revised implementation of the best play strategy.

```
g-rule +!win("bestplay", [int t, int h]) <: false {
  body {
    !setup_tactic(t);
  }

  rule +!setup_tactic(0) {}
  rule +!setup_tactic(int t2) {
    list hist = [];

    // construct a random history
    int i=0;
    while (i<h) {
      hist = hist + [M.randomInt() % 2];
      i++;
    }

    +strategy(t2, hist, M.randomInt() % 2);
    +score(t2, 0);

    !setup_tactic(t2-1);
  }

  rule $A.event("play", []) {
    list history = A.results();

    int max_len = -1;
int max_choice = -1;
int max_score = -1;
foreach (strategy(int s, list hist, int c) & score(s, int sc)) {
  int len = A.match(history, hist);
      if ((len > max_len) | ((len == max_len) & (max_score < sc))) {
        max_choice = c;
        max_score = sc;
          max_len = len;
        }
      }

      A.bid(max_choice);
    }

    rule $A.event("winner", [int bid]) {
  foreach (strategy(int s, list hist, bid) & score(s, int sc)) {
        -+score(s, sc+1);
      }
    }
  }
```

Explain this code a bid...

5.2 Preliminary Analysis

So, I have used the minority game example to try to get some timings for the impact of performance. I did this by reimplementing the player in ASER and leaving the rest the same.

I have done around 5 runs with each language. For each run, I have used the ASTRA main/compere agents and have used 29 player agents competing over 1000 rounds. All use the best-play strategy with the same history/strategy settings (see the cited paper if you want to understand the strategy better). I have then done some basic analysis on the results:

ASTRA averages around 0.004ms per cycle (but uses only around 50,000 cycles due to a dynamic scheduler that suspends the agent when the event+intention queues are empty).

ASER averages around 0.002ms per cycle (but uses around 500,000 cycles because the dynamic scheduler does not work for ASER).

This means that per cycle, ASER is approx 5 times slower than ASTRA in this example

That said, I also captured the overall duration of the game. There was no impact on this - both take around 19 seconds to complete

I can do this up as an experiment, with 5 runs per system, and calculate the average time per run / overall + the average number of cycles per run / overall

6 CONCLUSION