AgentSpeak(ER): Enhanced Encapsulation in Agent Plans

Alessandro Ricci[1], Rafael H. Bordini[2], Jomi F. Hübner[3], and Rem Collier[4]

[1] DISI, University of Bologna, `a.ricci@unibo.it`
[2] POLI-PUCRS, `r.bordini@pucrs.br`
[3] Federal University of Santa Catarina, `jomi.hubner@ufsc.br`
[4] University College of Dublin, `rem.collier@ucd.ie`

Abstract. In this paper, we introduce AgentSpeak(ER), an extension of the AgentSpeak(L) language tailored to support encapsulation. The AgentSpeak(ER) extension 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 based on AgentSpeak, illustrates the features of the language through examples, and discuss results of a case study based on the implementation of the proposed language.

1 Introduction

AgentSpeak(L) has been introduced in [25] with the purpose of defining an expressive, abstract language capturing the main aspects of the Belief-Desire-Intention architecture [7,16], 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) [19] 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 [2,6] and ASTRA [12]. Besides Agent Programming Languages, the AgentSpeak(L) model has been adopted as the main reference to development several BDI agent-based frameworks and technologies [3,4] as well as serving as inspiration for theoretical work aiming to formalise aspects of BDI agents and agent programming languages [30,14,5].

Existing Agent Programming Languages extended the language with constructs and mechanisms making it practical from a programming point of view [2]. Besides, proposals in literature extended the model in order to make it effective for specific kinds of systems — e.g. real-time systems [29] – or to improve the structure of programs, e.g. in terms of modularity [20,23].

Along this line, in this paper we describe a novel extension of the AgentSpeak(L) language — called AgentSpeak(ER)[5]— featuring *plan encapsulation*, i.e. the possibility

[5] The ER suffix stands for "Encapsulated Reactivity"

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:

- improving the overall readability of the agent source code, reducing fragmentation and increasing modularity;
- promoting a more goal-oriented programming style, enforcing yet preserving the possibility to specify purely reactive behaviour, properly encapsulated into plans for goals;
- improving intention management, enforcing a one-to-one relation between intentions and goals — so every intention is related to a (top-level) goal;
- improving failure handling, in particular simplifying the management of failures related to plans that react to environment events.

A first prototype implementation of the new language has been developed on top of Jason and ASTRA.

The remainder of the paper is organised as follows: first we describe in details the motivations that lead to the proposal of a new AgentSpeak extension (Section 2). We then introduce AgentSpeak(ER), defining the main concepts, syntax and informal semantics (Section 3). A case study about the implementation of the Minority Game is then used to discuss the approach (Section 4). Finally we conclude the paper discussing related work (Section 5) and sketching future work (Section 6).

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.

In the BDI model, plans are meant to specify some means by which an agent can satisfy an end [25]. 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. In the following, we refer to plans triggered by event goals as *g-plans*, and plans triggered by belief change (including percepts) as *e-plans*. The context specifies those beliefs that should hold given the agent's current belief base if the plan is to be triggered. The body of a plan is a sequence of actions or (sub-)goals.

In this approach — as well as in planning, in general — the *means* to achieve a goal (i.e., the plan 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 practice, when programming such systems, it is often the case that the strategy (the means) adopted to achieve some goal (the end) naturally includes reactions — i.e., reacting to events asynchronously perceived from the environment, including changes in the beliefs. This reflects more than just the ability of an agent to change/adapt its course of actions; it allows the integration of reactivity as a core ingredient of the strategy to achieve some goal. This revised notion of a plan is not just a programming feature; it also occurs naturally in human activity. For example, a fisherman with the goal of catching fish waits for the event of a tug on their line indicating a fish is on the hook. 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* reactions that are part of the strategy to achieve the goal, as well as the subgoals that are specific to that particular goal.

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 [25]. 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 waste at its location, then it can pick it up and bring it to the bin. This e-plan is an essential brick of the overall strategy to achieve the goal of cleaning the environment. The problem here is that it is an implicit rather than explicit goal of the agent (since it is an e-plan, it is executed regardless of the agent currently having the goal of keeping the environment clean). In practice, we adopt a maintenance goal [15] to clean the environment, which includes reacting to cleaning up waste when we see it. In the above program, this notion cannot be represented and remains in the mind of the programmer/designer; as there is no g-plan for it, there is no explicit trace in the agent mental structures about this goal.

This problem can also be illustrated with the following scenario. Consider an agent that includes a set of plans (a module written by a third party) to handle social obligations. The module has several e-plans for different types of obligations:

```
+obligation(Ag,committed(Goal)) : .my_name(Ag)      <- ...
+obligation(Ag,achieve(Goal))   : .my_name(Ag)      <- ...
-obligation(Ag,Goal) : .my_name(Ag) & .intend(Goal) <- ...
```

If for some reason during its execution an agent decides not to follow these plans anymore (e.g. it chooses to become disobedient), it is difficult to “disable” the behaviour of the above plans. Either these plans have to be changed to consider a particular state of the agent or the agent removes all these plans from its plan library. Neither option is simple to program. Although we can solve the problem, the lack of an explicit goal stating that the agent intends to be obedient is the cause of this problem.

Besides maintenance goal, also for achievement goals we see benefits in encapsulating the reactive behaviour in the corresponding plans. Let’s consider, as a very simple example, yet capturing the point, the goal of printing down all the numbers between N and 1, stopping if/when a ’stop’ percept is perceived. In AgentSpeak(L) 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).
```

The action `println` is meant to print the number on standard output. Here we exploit the fact that 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. More generally, in AgentSpeak(L) 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):

```
+!print_nums(0).
+!print_nums(N) : stop.
+!print_nums(N) : not stop <- !elab(N); !print_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 AgentSpeak(L) and in BDI architectures an e-plan can be introduced, using e.g. internal actions to act on the current ongoing intention:

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

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

Finally, the encapsulation of also impacts 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. 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 to each other.

Implicit vs. Explicit goals: Some parts of an agent program may have plans for which the goal is implicit.

Difficult failure handling: Failures/errors generated in the body of e-plan cannot be directly captured.

Besides the pure programming perspective, it is worth noting that these issues can affect the software engineering process for agent development. In particular, at the design perspective, 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 plans beyond the simple sequence of actions and goals, so as to include also the possibility to specify reactive behaviour encapsulated within the plan. Coherently, with the AgentSpeak(L) model, such a behaviour can be expressed in terms of e-plans. Accordingly, a plan becomes the *scope* of (*i*) a sequence of actions (referred as *body actions*), (*ii*) a set of *e-plans*, specifying a reactive behaviour which is active at runtime only when the plan is in execution, and (*iii*) a set of *g-plans*, specifying plans to achieve subgoals that are relevant only in the scope of this plan. The e-plans and g-plans are referred to as *sub-plans*. The sub-plans may include also reactions to failures occurring when the plan is executed.

The second change is enforcing that e-plans — as pieces of reactive behaviours — must always be defined in the context of a g-plan. We are thus enforcing the principle that an agent does (and reacts) always because of a goal to achieve or maintain. This ensures that programmers explicitly specify what is the goal to achieve even when defining a purely reactive behaviour. In so doing, at runtime every intention[6] has an associated goal being pursued.

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 exemplified below:

```
/*  g-plans to achieve goal g in context c */
```

[6] As in AgentSpeak(L), an intention is the result of the deliberation to commit to some desire. Briefly, if the agent has an applicable plan for a goal event (i.e. a desire), it commits to it by creating an intention based on that plan and starts executing it.

```
+!g : c <: gc {

  <- a; b; ?g1; !g1; !!g2. // plan body (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 with a head and a body. Besides the triggering event and the context, in AgentSpeak(ER) the head has a third new element: a *goal condition*, optionally written after `<:`, with the same syntax as the context. While the context is a pre-condition to select a plan as applicable for an event, the goal condition is a post-condition that defines when the goal can be considered as achieved[7]. Any goal created based on this g-plan is considered achieved if and only if this condition holds. If no goal condition is specified, the goal is considered achieved as soon as the body execution completes, as usual in AgentSpeak(L). If the case that no plan body is specified as well, then the goal condition is considered `false`—it is the case of never-ending maintenance goal. However, if a goal condition is defined, having finished the body execution is not sufficient to deactivate the goal. Notice that if the goal condition becomes true while the body actions are being executed, the execution ceases immediately, since the goal being achieved means no further action would be necessary.

In AgentSpeak(ER) the body defines the g-plan scope enclosed by '`{`' and '`}`' and is composed of the body actions (after '`<-`' and before '`.`') as well as sub-plans. Like in AgentSpeak(L), as soon as a g-plan is instantiated, the body actions start to execute. The body actions can be empty — this is the case of g-plans expressing a purely reactive behaviour. The sub-plans of the g-plan are considered as relevant only for events produced by the g-plan execution, since they are in the scope of the g-plan.

While a g-plan is executing, it can be *interrupted* by events relevant for its e-plans. When the agent perceives an event and an e-plan from g-plan is applicable according to its specified context, then the execution of the g-plan body is suspended until the body of the e-plan finishes its execution. In AgentSpeak(L), other plans do also interrupt

[7] Or considered impossible to achieve, or the motivation for the goal no longer holds, etc. Programmers can use this for any condition that implies the goal should no longer be pursued. Note that, when programming declaratively, this condition is likely to include the goal in the triggering event itself (as believed to be true).

the execution of a plan in the case of subgoals. For example, in the body "a1; a2; !g1; a3", g1 is a sub-goal and thus the action a3 is executed after g1 is achieved. The plan body execution in interrupted synchronously. In AgentSpeak(ER) this uniformly occurs also with any kind of events, not only sub-goal events; that is, the body actions can be interrupted to react to events coming from the environment. However, in this case the interruption is *asynchronous* – the point where the body actions is interrupted is unknown, and depends on the environment, at runtime.

3.2 Examples

To give a more concrete taste of the language, we consider again the examples seen in Section 2, now rewritten in AgentSpeak(ER). The robot cleaning example becomes:

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

We can give an explicit reason for the reactive behaviour by encapsulating the e-plan inside a g-plan, with an explicit goal `clean_env`. Setting the goal condition to `false` means that the plan execution is going to last forever. It is a way of implementing some form of maintenance goal. This is also a particular case where the body of the g-plan does not have any actions.

The `print_nums` example seen before can be rewritten to fully encapsulate the strategy in the same g-plan:

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

The goal `print_nums` is achieved either by the perception of `stop` or `done` by the execution of its body actions.In case we want to take some action to react to `stop`, we can introduce an e-plan as follows:

```
+!print_nums(0).
+!print_nums(N) <: done {
   <- !elab(N); !print_nums(N-1).
   +!elab(N) <- ...
   +stop <- println("stopped"); +done.
}
```

For the example of plans to handle social obligations, we can define an explicit goal by encapsulating the e-plans in a g-plan as follows:

```
+!be_obedient {
   +obligation(Ag,committed(Goal)) : .my_name(Ag)      <- ...
   +obligation(Ag,achieve(Goal))   : .my_name(Ag)      <- ...
   -obligation(Ag,Goal) : .my_name(Ag) & .intend(Goal) <- ...
}
```

Now the agent can disable these plans by simply performing `.drop_intention(be_obedient)`, and resuming its obedient behaviour by adopting the goal `!be_obedient`, and it can also check whether it is being obedient by testing `.intend(be_obedient)`.

With the new language, we can program different kinds of commitments to goals [11,31] by exploiting the goal condition. For instance, the Single Minded Commitment for goal `g` can be programmed as:

```
+!g <: g | f {
   ...
}
```

where `f` states when the goal becomes impossible. The agent commits to achieve `g` until `g` is believed either true or impossible to achieve. To program this kind of commitment in AgentSpeak(L), we have to follow some programming patterns requiring extra plans (three extra plans as shown in [18]).

3.3 Failure Management

Failures generated by the execution of actions of sub-plans for a goal `g` can be handled and managed by:

- subplans of type `-!g` listed in the body of the g-plan;
- plans at the same level of `+!g`, if the event is not managed within the body.

In terms of the new language style, we can keep the failure plans within the scope of the g-plan. However, it is also reasonable that if we define a plan for `+!g` at a certain level of a g-plan tree[8], it might make sense for the programmer if the plans for `-!g` are all placed at the same level (specially if the programmer is influenced by the style of the Jason variant of AgentSpeak).

3.4 Key points

We conclude this section by summarising the key points brought out by AgentSpeak(ER):

Encapsulation – The strategy to achieve a goal is encapsulated in one or multiple g-plans, each embedding also the reactive behaviour which is part of the strategy. The effect is to reduce code fragmentation, improving its understanding.

Explicit goals – Every behaviour of the agent is now explicitly related to some goal to achieve. This promotes a more goal-oriented programming style, yet preserving the possibility to easily define g-plans based on purely reactive strategies; this allows the agent to better *manage* its intentions. For instance, a programmer can now do a simple `.drop_intention(g)` to disable the behavior of e-plans embedded in the corresponding g-plans. In AgentSpeak(L), the relation goal-intention is not

[8] Note that g-plans within g-plans now implicitly form a plan tree for a top-level goal, and the plan library is thus a forest of such trees.

one-to-one. AgentSpeak(L) can have intentions from e-plans and so intentions without a explicitly represented goal. In AgentSpeak(ER) there is a one-to-one relation between goals and intentions. All intentions come from goals (only explicit goals are allowed). Primitives to handle goals can thus handle *all* intentions. In AgentSpeak(L), primitives to handle goals can manage a limited set of the intentions (those created from goal addition plans).

Failure handling – Thanks to explicit goals, failures generated in the body of an e-plan can now be directly captured and managed by failure recovery plans defined for g-plan, without the need for auxiliary goals and plans to be introduced.

Coarse-grained intentions – In AgentSpeak(L), each e-plan in execution has its own intention/stack, which runs concurrently to the other intentions. Conceptually, this follows the idea that the management of environment events are not part of an existing overall plan to achieve some goal. In the new model instead, an e-plan inside a g-plan is meant to specify a behaviour that is useful for achieving the goal of the g-plan, so part of the same intention. For this reason, if an e-plan (sub-plan) 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 then to more coarse-grained intentions.

As a final remark, the choice of constraining the definition of e-plans to the scope of a g-plans deserves some further clarification. Apparently, it may seems as a limitation, making the approach not capable of capturing purely reactive behaviour, forcing a designer to define artificial goals. However, in software development a designer has always a goal in mind when writing down the behaviour of an agent: this is true also for purely reactive agents. This is different if we consider e.g., agent-based modelling and simulation, where the definition of the behaviour of an agent could be driven by the description of a local behaviour. The extension presented in this paper is explicitly targeted to the design and programming of agents as software components responsible of autonomously performing some tasks. The extension then promotes a better discipline, enforcing the designer to always provide an explicit representation (at the programming level) about the goals she/he has in mind.

4 First Implementation and Discussion

In order to play with the new language and do a first evaluation, a first prototype implementation has been developed on top of both Jason [2] and ASTRA [12][9] .

The Jason prototype has been used to implement the examples used in this paper. The ASTRA prototype uses a syntax slightly different than that used by examples provided in Section 3, being it adapted to the syntactic style adopted in ASTRA.An example follows:

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

[9] The Jason extension is available here: `https://github.com/agentspeakers/jason-er`

```
  }

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

The different syntax however does not impact the associated semantics.

In the remainder of this section we show the benefits of AgentSpeak(ER) by considering an existing program, rewritten with the new language. ASTRA is adopted as implementation language—however the same discussion would apply by considering Jason as implementation language.

4.1 Minority Game

We have adapted a simulation of the Minority Game (MG), a well-known model for evaluating collective behaviour of agents competing for finite resources [21]. 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 areas such as modelling financial markets [9] and traffic simulation [10].

The existing implementation (see Figure 1) consists of 3 types of agent: the *compere* agent, who is responsible for managing the game (starting rounds, evaluating bids, ...); the *player* agents, who implement a set of MG strategies; and the *main* agent, which is responsible for configuring the game (creating and configuring the compere and player agents). Interaction between the compere and the players is through a shared game-board artifact implemented using CArtAgO [26].

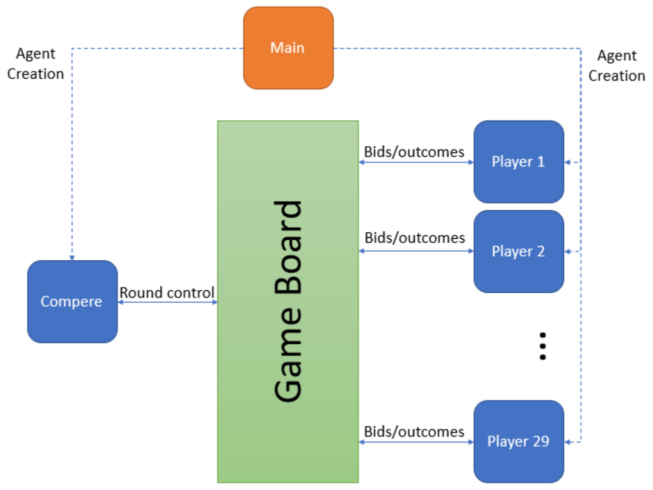

Fig. 1: Minority Game Agent Architecture

The existing implementation currently consists of 2 types of plan: *configuration plans*, one of which is called when the agent is created; and *strategy plans* which implement the various potential strategies that the agent can use. A subset of the ASTRA implementation of the *player* agent is given below:

```
rule +!main(["bestplay", [int t, int h]]) {
  -+strategy("bestplay"); !setup_tactic(t, h); !setup();
}
rule +!main([string strategy, []]) {
  -+strategy(strategy); !setup();
}
rule +!setup_tactic(0,int h) {}
rule +!setup_tactic(int t2, int h) {
  list hist = [];
  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, h);
}

rule $cartago.signal(string id, play()) : strategy("bestplay") {
  cartago.results(java.util.ArrayList history);

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

rule $cartago.signal(string id, winner(int bid)) : strategy("bestplay") {
  foreach (strategy(int s, list hist, bid) & score(s, int sc)) {
    -score(s, sc); +score(s, sc+1);
  }
}
```

In the above code, the `!main(...)` plans are the configuration plans. The `!setup_tactic(...)` plans are part of the bestplay strategy but are also used for configuration. This is because, unlike simpler strategies such as *Random Bid* and *Tit-for-Tat*, the best play strategy requires that a set of random strategies be created as part of its configuration. In best play, a random strategy is a randomly generated set of `h` outcomes combined with a recommended next move. Each strategy has an associated score which is used to help determine how successful each strategy is.

The strategy plans are the `$cartago.signal(...)` plans, which handle the `play()` and `winner(...)` signals respectively. These signals are generated by the game board when the *compere* agent performs operations on it (e.g. when a round starts/ends). For bestplay, the `play()` signal triggers a behaviour where the agent compares the game outcome history against its strategies and picks the strategy that best fits the history (longest matching subsequence). The corresponding next move is played (via the `bid()` artifact operation). Conversely, the `winner(...)` signal triggers a behaviour where the agent updating the score for all strategies that lead to `bid` being selected.

Some of the key points to note when reviewing the above code are: (i) a custom plan is needed to handle the `!main(...)` goal in the case of the bestplay strategy because it requires some custom initialisation; (ii) a strategy belief is required to enable the identification of the plans that are relevant to the selected strategy; and (iii) there is no guarantee that these plans will be grouped together in the implemented agent. They could be spread throughout the agents codebase making it difficult to read and understand the overall behaviour.

In the AgentSpeak(ER) implementation, we still maintain the two types of plan, however instead on needing multiple plans to capture a strategy, the entire strategy is now encapsulated within a single g-plan (of course the other plans still exist as subplans of the g-plan). The snippet of code below contains the AgentSpeak(ER) code to the ASTRA example given above:

```
rule +!main([string strategy, list config]) {
  !win(strategy, config);
}

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

  rule +!setup_tactic(0) {}
  rule +!setup_tactic(int t2) {
    list hist = [];
    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 $cartago.signal(string id, play()) {
    cartago.results(java.util.ArrayList history);

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

  rule $cartago.signal(string id, winner(int bid)) {
    foreach (strategy(int s, list hist, bid) & score(s, int sc)) {
      -score(s, sc); +score(s, sc+1);
    }
  }
}
```

As can be seen from the above snippet of code. The implementation of the two types of agent is quite similar. In fact many of the rule implementations have not changed significantly. However, there are a few interesting observations regarding the revised implementation: *(i)* the complexity of the plan contexts were simplified when using g-plans because the g-plan itself provided some of the context; *(ii)* the number of arguments passed as parameters was reduced, again because the scope of the parameters of the g-plan was the plan body *and* all of its sub-plans; *(iii)* the total number of rules under consideration on each iteration was significantly less because only the rules within an active g-plan were considered by the agent. This means that, when an agent has multiple strategies, only the rules relating to the active strategy will be considered, whereas in ASTRA, all rules are always considered.

4.2 A Note on Performance

After reviewing our new language, it became apparent that (i) it was reducing the number of plans that need to be evaluated on each iteration; but (ii) the introduction of a goal condition introduces a significant new overhead because it must be evaluated on each iteration. As a result, it was decided that a comparison of the new and old languages be carried out. Initially, we compared interpreter cycle execution time and the number of iterations based on a single configuration of the MG with 29 players and 1000 rounds. For our results, we averaged the values across all 29 players and repeated the experiment 5 times.

Table 1: Comparing ASTRA and AgentSpeak(ER)

	ASER	ASTRA
Cycle Time (ms)	0.0017	0.0036
Cycles	293,772	62,880
Elapsed Execution Time (ms)	495.22	229.29
Unix (timed)	16s	15s

Results of our initial comparison can be found in Table 1. The difference in the number of cycles is due primarily to the scheduling algorithm used by ASTRA, which suspends agents that have no sensors (perceptors) and whose event and intention queues are empty. The impact of this is that ASTRA is generally more efficient than AgentSpeak(ER). Due to the small but consistent difference in performance between the unix timing of the two experiments, we then explored how increasing the number of rounds affected performance. Results for this are shown in Table 2.

Table 2: ASTRA vs AgentSpeak(ER) Performance

	1000	2000	3000	4000	10000
ASER (s)	15.514	30.251	44.202	57.736	140.059
ASTRA (s)	14.893	28.232	42.168	56.453	137.379
Diff. (%)	4.2%	6.8%	4.8%	2.3%	2.0%

This second table shows that the introduction of g-rules in AgentSpeak(ER) has only a small impact on performance. Here, it is almost linear. Further, ASTRA shows a marginal performance improvement of between 2-6%.

While this is not intended to be a thorough evaluation of AgentSpeak(ER), it is useful because it hints that the use of goal conditions does not significantly impact the performance of the language. It must also be noted that the prototype implementation is not as mature as the ASTRA implementation — interpreter optimisations could further reduce the difference in performance.

5 Related Work

AgentSpeak(ER) is primarily related to work in literature focusing on improving cognitive BDI agent programming. A main aspect widely discussed and developed in the literature is *modularity* [20,8,13,22,24,27,17,23]. In programming languages, modularity is strongly related to encapsulation. In fact, strengthening encapsulation typically leads to refining modularity, in particular devising more coarse-grained modules. AgentSpeak(ER) enriches the spectrum of approaches elaborated in literature for improving modularity in BDI-based agent programming languages by devising coarse-grained plans as modules encapsulating goal-oriented *and* reactive behaviour.

Besides, AgentSpeak(ER) is related to existing BDI agent programming languages extending the basic plan model as found in the original proposal of AgentSpeak(L). In this context, a main reference is CANPlan [28], a BDI-style agent-oriented programming language enhancing usual BDI programming style with declarative goals, look-ahead planning, and failure handling. It allows programmers to mix both procedural and declarative aspects of goals, enabling reasoning about properties of goals and decoupling plans from what these plans are meant to achieve. The lookahead planning makes it possible to guarantee goal achievability and avoid undesirable situations. The plan model adopted in CANPlan is analogous to the AgentSpeak(L) one. Each plan is characterised by a plan rule $\mathsf{e(t)} : \psi(\mathsf{xt, y}) \leftarrow \mathsf{P(xt, y, z)}$., where P is a “reasonable strategy” to follow when ψ is believed true in order to resolve/achieve the event. P can be a rich composition of actions but not reactions. Reactive behaviours can be expressed instead — like in AgentSpeak(L) and in the basic BDI — as separate plans handling belief updates corresponding to environment events.

6 Conclusion

In this paper, we introduced AgentSpeak(ER), a novel extension of the classical AgentSpeak(L) language. The language provides encapsulation for agent goals, which clearly improves legibility and reusability of AgentSpeak code. Furthermore, the new language improves some of the shortcomings of AgentSpeak in regards to goal orientation and declarative goals by ensuring that all reactive plans are also associated with general goals, providing a "goal condition" which means goals can be still active even though presently there is no action for the agent to take towards that goal, and allowing external events (i.e. reactions to changes in beliefs) to trigger various plans, for all the goals it might be relevant. The proposal was implemented and experimentally evaluated on top of the ASTRA platform.

As with any new programming language, there is much future work, some in fact ongoing. We are currently refining the ASTRA implementation, trying to make a few optimisations to improve the evaluation results we reported in this paper. A Jason-based implementation is also under way; comparison of the performances of the two implementations might lead to insights that might improve the implementation of the platforms themselves.

More generally, full understanding and evaluation of a programming language takes many years. We expect in the long term to use AgentSpeak(ER) in the practical development of multi-agent systems, both for real-world systems and also academic ones (e.g., for the multi-agent programming contest [1]). However, besides the actual programming practice, we expect AgentSpeak(ER) to contribute to formal work as well. Assessing how formal verification of AgentSpeak(ER) systems compares to the original language is also planned as future work.

References

1. Ahlbrecht, T., Fiekas, N., Dix, J.: Multi-agent programming contest 2016. International Journal of Agent Oriented Software Engineering (2018), to appear
2. Bordini, R., Hübner, J.: BDI agent programming in AgentSpeak using Jason. In: Toni, F., Torroni, P. (eds.) CLIMA VI, LNAI, vol. 3900, pp. 143–164. Springer (Mar 2006)
3. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. No. 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer (2005)
4. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer (2009)
5. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous Agents and Multi-Agent Systems 12(2), 239–256 (2006), https://doi.org/10.1007/s10458-006-5955-7

6. Bordini, R.H., Hübner, J.F., Wooldrige, M.: Programming Multi-Agent Systems in AgentSpeak using *Jason*. Wiley Series in Agent Technology, John Wiley & Sons (2007), `http://jason.sf.net/jBook`
7. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Computational Intelligence 4, 349–355 (1988), `http://dx.doi.org/10.1111/j.1467-8640.1988.tb00284.x`
8. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI Agents in Functional Clusters, pp. 277–289. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
9. Challet, D., Marsili, M., Zhang, Y.C., et al.: Minority games: interacting agents in financial markets. OUP Catalogue (2013)
10. Chmura, T., Pitz, T.: Minority game: Experiments and simulations of traffic scenarios. Tech. rep., Bonn econ discussion papers (2004)
11. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artificial Intelligence 42, 213–261 (1990)
12. Collier, R.W., Russell, S.E., Lillis, D.: Reflecting on agent programming with agentspeak(l). In: Chen, Q., Torroni, P., Villata, S., Hsu, J.Y., Omicini, A. (eds.) PRIMA 2015: Principles and Practice of Multi-Agent Systems - 18th International Conference, Bertinoro, Italy, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9387, pp. 351–366. Springer (2015)
13. Dastani, M., Steunebrink, B.: Modularity in BDI-based multi-agent programming languages. In: 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology. vol. 2, pp. 581–584 (Sept 2009)
14. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A common semantic basis for BDI languages. In: Dastani, M., Fallah-Seghrouchni, A.E., Ricci, A., Winikoff, M. (eds.) Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, USA, 2007, Revised and Invited Papers. Lecture Notes in Computer Science, vol. 4908, pp. 124–139. Springer (2007), `https://doi.org/10.1007/978-3-540-79043-3_8`
15. Duff, S., Harland, J., Thangarajah, J.: On proactivity and maintenance goals. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 1033–1040. AAMAS '06, ACM, New York, NY, USA (2006), `http://doi.acm.org/10.1145/1160633.1160817`
16. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2. pp. 677–682. AAAI'87, AAAI Press (1987), `http://dl.acm.org/citation.cfm?id=1863766.1863818`
17. Hindriks, K.: Modules as Policy-Based Intentions: Modular Agent Programming in GOAL, pp. 156–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
18. Hübner, J.F., Bordini, R.H., Wooldridge, M.: Programming declarative goals using plan patterns. In: Baldoni, M., Endriss, U. (eds.) Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies (DALT 2006), held with AAMAS 2006, 8th May, Hakodate, Japan. LNCS, vol. 4327, pp. 123–140. Springer (2006)
19. Ingrand, F.F., Georgeff, M.P., Rao, A.S.: An architecture for real-time reasoning and system control. IEEE Expert: Intelligent Systems and Their Applications 7(6), 34–44 (Dec 1992), `http://dx.doi.org/10.1109/64.180407`
20. Madden, N., Logan, B.: Modularity and Compositionality in Jason, pp. 237–253. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Moro, E.: The minority game: an introductory guide. arXiv preprint cond-mat/0402651 (2004)
22. Novák, P., Dix, J.: Modular bdi architecture. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 1009–1015. AAMAS '06, ACM, New York, NY, USA (2006), `http://doi.acm.org/10.1145/1160633.1160814`
23. Nunes, I.: Improving the Design and Modularity of BDI Agents with Capability Relationships, pp. 58–80. Springer International Publishing, Cham (2014)

24. Ortiz-Hernández, G., Hübner, J.F., Bordini, R.H., Guerra-Hernández, A., Hoyos-Rivera, G.J., Cruz-Ramírez, N.: A Namespace Approach for Modularity in BDI Programming Languages, pp. 117–135. Springer International Publishing, Cham (2016)
25. Rao, A.S.: Agentspeak(l): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996)
26. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Tools and Applications, chap. 8, pp. 259–288. Springer (2009)
27. van Riemsdijk, M.B., Dastani, M., Meyer, J.J.C., de Boer, F.S.: Goal-oriented modularity in agent programming. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 1271–1278. AAMAS '06, ACM, New York, NY, USA (2006)
28. Sardina, S., Padgham, L.: A bdi agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems 23(1), 18–70 (Jul 2011)
29. Vikhorev, K., Alechina, N., Logan, B.: Agent programming with priorities and deadlines. In: The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1. pp. 397–404. AAMAS '11, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2011), `http://dl.acm.org/citation.cfm?id=2030470.2030529`
30. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M. (eds.) Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002. pp. 470–481. Morgan Kaufmann (2002)
31. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (2002)