A Methodology to Extend Imperative Languages with AgentSpeak Declarative Constructs

Loris Fichera

University of Catania – Engineering Faculty Viale Andrea Doria, 6 — 95125 - Catania, ITALY EMail: loris.fichera@gmail.com

Daniele Marletta Scuola Superiore di Catania Via S. Nullo 5/i — 95123 – Catania, ITALY EMail: damarletta@ssc.unict.it

Vincenzo Nicosia Scuola Superiore di Catania – Laboratorio sui Sistemi Complessi Via S. Nullo 5/i — 95123 – Catania, ITALY EMail: vincenzo.nicosia@ct.infn.it

Corrado Santoro University of Catania – Dept. of Mathematics and Computer Science Viale Andrea Doria, 6 — 95125 - Catania, ITALY EMail: santoro@dmi.unict.it

Abstract—This paper presents a novel technique for the support of declarative constructs of AgentSpeak in a imperative language. Starting from an analysis of the characteristics and semantics of AgentSpeak, as well as the requirements of an AgentSpeak implementation, the paper proposes a framework which, by exploiting object-orientation and operator overloading, is able to enrich an imperative language with declarative constructs suitable to represent and write AgentSpeak programs. A concrete implementation of the proposed framework is then given by presenting *PROFETA* (*Python RObotic Framework for dEsigning sTrAtegies*), a tool written in Python which adds to this language the capability to write and execute AgentSpeak plans. A case-study shows how PROFETA can be used to easily write strategies for an autonomous mobile robot, designed by the authors for the "Eurobot 2010" international robotic competition.

I. INTRODUCTION

Software agents are an abstraction for autonomous entities which have a certain amount of knowledge, interact with an environment and play a given role in order to achieve a goal. They have been used in the last decade as a reference model for the solution of a variety of problems in different fields, from automatic reasoning to data mining, from search in semantic web to simulation of social behaviours. The adaptability of agent–oriented modelling and the possibility to represent different problems as agent–based systems have produced a proliferation of agent models, such as reflex– based agents, goal–based agents, utility–based agents, learning agents [3], [20], [14], [16], each of them with peculiar characteristics which make it useful for the solution of a specific class of problems. At the same time many agent– oriented programming languages (Soar [21], NetLogo [17], 3APL [6], GOAL [7]) and agent–based platforms (JADE [1], MASON [10], Repast [18], Swarm [19]) have been developed in the last decade, in order to give programmers powerful and easy–to–use tools to model, develop and maintain agent–based software.

An increasing interest has been recently devoted to exploit goal-based agent models for the realization of intelligent autonomous systems [11], [15]. One of the proposed paradigms for goal-oriented agents is the Belief-Desire-Intention (BDI) [13], which models the knowledge as a set of beliefs about environment and internal state of the agent, then represents as *desires* the goals to be achieved, which expand into a set of *intentions*, i.e. atomic actions to be performed in order to achieve a desire. There exists a reference abstract language for BDI agents, which is called AgentSpeak(L) [12], and defines a declarative syntax to express desires and intentions in a way which is much similar to rule-based logic programming: beliefs are treated as *facts* while desires and intentions respectively correspond to conditions and body of rules. The only existing implementation of AgentSpeak semantic is Jason [9], which is an interpreter, written in Java, for an extended AgentSpeak syntax written. Jason lets the programmer write its AgentSpeak statements in a separate file and requires actions and event handlers to be implemented by Java classes. Unfortunately, all the AgentSpeak program is completely separated from action implementation, while it

ag	::=	bs ps	
bs	::=	$at_1,\ldots,at_n.$	$(n \ge 0)$
at	::=	$P(t_1,\ldots,t_n)$	$(n \ge 0)$
ps	::=	$p_1,\ldots,p_n.$	$(n \ge 1)$
p	::=	te:cd < - h.	
te	::=	$+at \mid -at \mid +g \mid -g$	
cd	::=	$\texttt{true} \mid l_1 \And \ldots \And l_n$	$(n \ge 1)$
h	::=	$\texttt{true} \mid f_1 \hspace{0.1 in}; \hspace{0.1 in} \ldots \hspace{0.1 in}; \hspace{0.1 in} f_n$	$(n \ge 1)$
l	::=	$at \mid \texttt{not} at$	
f	::=	$\mathtt{A}(t_1,\ldots,t_n) \mid g \mid u$	$(n \ge 0)$
g	::=	$!at \mid ?at$	
u	::=	$+at \mid -at$	

Fig. 1. Basic AgentSpeak Syntax

would be very useful to preserve a declarative AgentSpeak model *into* the same programming environment.

This paper shows that it is possible to embed an AgentSpeak semantic into any imperative language which allow operator overloading, and proposes an example of such embedding in Python. The developed tool is called *PROFETA* (for *Python RObotic Framework for dEsigning sTrAtegies*), and has been successfully used to implement agent–based intelligent autonomous systems to drive robots in dynamic environments [5].

The paper is organised as follows. Section II provides an overview of AgentSpeak(L) syntax and semantics. Section III explains the methodology used to embed AgentSpeak(L) semantic into imperative languages through operator overloading. Section IV describes the implementation of PROFETA. Section V illustrates a case-study of PROFETA in a real application. Section VI reports conclusions and future works.

II. AGENTSPEAK: REQUIREMENTS FOR ITS IMPLEMENTATION

A. Overview of AgentSpeak

AgentSpeak(L) [13], [12] (or simply AgentSpeak) is an abstract declarative language designed many years ago to program BDI agents. According to its basic syntax, first introduced in [13], [12] and then extended in [2], which is reported in Figure 1, an agent program is made of a set of *beliefs* and a set of *plans*.

Beliefs represent the *knowledge* of the agent and are expressed using atomic formulae made of a predicate symbol P and a set of zero or more *ground terms*. The same syntax (predicate with ground terms) is also used to represent *goals*, which are the AgentSpeak abstractions for desires. Two types of goals are provided: (*i*) achievement goal, expressed as !at, meaning that the agent wants to make true formula at (i.e., ensure that at is a known belief); and (*ii*) test goal, expressed as ?at, intended to verify whether formula at is true (i.e., check if at is a known belief).

Plans are the *tasks* that the agent has to execute and are expressed by means of rules in the form te : cd < -h; here:

- te is the event triggering the rule; it can be the addition or removal of a belief (+at or -at), or the request to achieve (or no more achieve) a goal (+!at or -!at);
- cd is the *condition*¹, that is made of a set of beliefs that must be all true in order to trigger the plan. The specified beliefs may contain ground terms or variables, and express the knowledge the agent must have when the event te fires in order to trigger the plan.
- *h* is a set of *atomic actions* which are executed, in sequence, following plan triggering. They include, once again, addition or removal of a belief (+*at* or -*at*), request to achieve or no more achieve a (sub-)goal (+!*at* or -!*at*), or the execution of a user-defined action. With the exception of the last case (user-defined action), all the other actions, when executed, generate the relevant trigger event causing the possible selection of another plan.

As the reader can understand, an AgentSpeak program is a set of rules in the form *event-condition-action(s)*; events may come from the perception of the agent's environment (in this case they are called *external events*) or be generated by an action (*internal events*), such as the achievement of a sub-goal. On the basis of event type, the way in which it is processed differs, according to the specific semantics of the language which is described in [12], [9], [4].

An example of an AgentSpeak program running in a mobile robot is given in Figure 2. Let us consider that the robot aims at grabbing two types of objects, balls and cylinders, and that, to accomplish this task, it has two different mechanical arms, one for each type of object. A camera, with a proper artificial vision software, is used to detect the presence of an object to be picked. Two predicates are used for beliefs: object seen, which is asserted by the artificial vision software when an object of a certain type is detected in a given (x, y) position, and arm deployed, representing which one of the two arms is currently in the deployed configuration (and thus able to pick the object of the same kind). According to the program in Figure 2, when the camera detects an object, the proper belief is asserted causing the first plan to be triggered and executed; this plan contains the achievement of three goals: first the proper arm has to be deployed, then the position of the detected object has to be reached so that finally the object can be picked. In achieving the first goal (prepare_arm), we check if an arm not able to pick the recognised object is currently deployed and, if this is the case, we retract the current arm and deploy the other one. We suppose that such retract/deploy actions, during their execution, change the belief set by properly updating the arm_deploy belief². The second goal aims at reaching the target position and, as the source code reports, triggers the actions needed to orientate the robot towards the target point and then to go to it. The last goal performs object picking by activating the corresponding mechanical arm.

¹AgentSpeak uses the term "context", but we prefer "condition" because it is more appropriate for the real meaning of the cd part of the plan.

²This is in accordance with AgentSpeak model.

```
+object_seen(Type, X, Y) : true <-
          +!prepare_arm(Type);
          +!reach object(X, Y);
          +!pick_object(Type).
+!prepare_arm(ball) :
     arm_deployed(cylinder) <-</pre>
          retract arm(cylinder);
          deploy_arm(ball).
+!prepare_arm(cylinder) :
     arm_deployed(ball) <-</pre>
          retract_arm(ball);
          deploy_arm(cylinder).
+!prepare_arm(X) :
     arm_deployed(X) <-</pre>
          true.
+!reach object(X,Y) : true <-
          orientate towards(X,Y),
          go_to(X,Y).
+!pick_object(X) : true <-
          activate arm(X).
```

Fig. 2. An object picking robot

B. From Theory to Practice: Architecture of an AgentSpeak Interpreter

AgentSpeak represents a very flexible language to express the behaviour of agents using a goal-oriented approach; in order to allow the development of such agents, a proper interpreter needs to be designed and implemented. To this aim, we could imagine three different design approaches: *totally declarative, a mix of declarative and imperative, totally imperative.*

The first approach, totally declarative, requires the extension of AgentSpeak with additional statements in order to derive a complete programming language. These statements include constructs to modify variables and to apply them mathematical/logical operators, as well as proper generalpurpose library functions to interact with the environment (i.e., sensors and actuators). Obviously, all the additional statements and constructs must obey to the declarative/goaloriented paradigm which is the basis of AgentSpeak; while this aim can be easily achieved for variable assignment and mathematical/logical operators (which could be, for instance, borrowed from Prolog), the same is not straightforward to be achieved for functions. After all, what is a "function" in AgentSpeak? Given that in imperative programming a "function" is the evolution of the concept of "subroutine", in a goal-oriented language like AgentSpeak the execution of a function can be viewed as the achievement of a "sub-goal".

Indeed, AgentSpeak semantics specifies that the presence of a goal achievement statement +!at in a plan implies to *suspend* the execution of the plan, *perform sub-goal achievement* and then *resume* the plan; such a behaviour is therefore quite similar to that of subroutine execution. Starting from these considerations, a totally declarative approach seems a viable solution, even if its validity, effectiveness and efficiency has to be verified. Nevertheless, extending AgentSpeak to make it a complete declarative language is beyond the scope of this paper.

The second approach, a mix of declarative and imperative paradigms, is the easiest solution and implies the use of an imperative language to build the interpreter of the AgentSpeak program and then write, using the imperative language itself, all the parts which cannot be implemented in AgentSpeak, i.e. specific agent actions and event catching and signalling. This is the approach exploited by Jason [9], the most widely known AgentSpeak implementation which is written in Java³. In Jason, agent's behaviour has to be written using AgentSpeak syntax (in a separate source file), while a set of specific classes, to be developed in Java, are required to implement all the atomic actions specified in the AgentSpeak file as well as the computations needed to poll and generate external events. While this approach is valid and effective, even if a bit naïve, it has some drawbacks. Indeed, there are two different "worlds", the declarative one with its syntax, semantics, objects and identifiers, and the imperative one, with a different syntax and semantics, somewhat "glued" to the declarative part in order to allow the access to and manipulation of the knowledge-base (beliefs). This is unfavourable for both program design and execution: the former aspect forces a programmer to handle different source files with different semantics while the latter requires proper transformations to pass data between the two domains which undoubtedly affect performances. Moreover, the AgentSpeak file cannot be compiled, thus impeding to deploy only binary files in a live system.

The third approach, totally imperative, at first sight sounds quite weird: how can we provide declarative semantics only using imperative constructs? Indeed, to execute an AgentSpeak program, a proper processing engine needs to be written, and this can be done in an imperative language⁴; the same engine could provide an API with proper function calls able to define beliefs, goals and plans by using the imperative language itself. Surely, a skilled software engineer can easily verify that such an approach is even worse than mixing imperative and declarative paradigms: the resulting source file could easily become unreadable and hard to maintain since the declarative semantics would be somewhat "hidden" in the API function calls. But if the language is object-oriented and supports operator overloading, such features can be exploited to solve the problems above and provide an "all-in-one" environment. This is described, in details, in the following Sections.

 $^{^{3}\}mbox{and}$ probably, the sole AgentSpeak implementation at the time this paper has been written.

⁴After all, the Jason engine is written in Java!

III. EMBEDDING DECLARATIVE CONSTRUCTS INTO AN IMPERATIVE LANGUAGE

Let us suppose we have an imperative language, let us call it *host language*, and let us consider that we want to have the possibility of writing AgentSpeak statements in the host language, without changing the compiler and the (standard) runtime library, and without introducing something like a preprocessor able to suitably perform a syntactic transformation of the declarative code⁵. The question to answer is: could we write (and execute), for example in C, something like this?

The answer is obviously "*No!*", since syntax and semantics are not in accordance with C rules. However, we could reformulate the question as follows: could we try to modify something, in the code above, in order to make it writable (and executable), in an imperative language such as C? Let's deal with such a problem by taking into account syntax first, and then semantics.

From the syntactical point of view, the construct above has three main problems, due to the presence of the symbols ":", "<-" and ".". Indeed, "+" and "!" are valid operators, and reach object(X,Y), true, orientate_towards(X,Y) and go_to(X,Y) are valid constructs (functions calls and symbol/variable evaluation). Therefore, let us reasonably replace non-recognisable symbols with other symbols which conform to C syntax. To this aim, since symbol ":" can be interpreted as "such that", a candidate replacement is "|"; in a similar way, symbol "<-", whose meaning is a sort of implication, can be replaced with "<<". The last symbol, the dot ".", is employed to signal the ending of the sequence of actions to be performed following plan activation; in this case, its replacement entails to find a proper way to represent a sequence or block of actions: the C block, i.e. $\{\dots\}$, is worth to be used in this case.

According to such modifications, the AgentSpeak plan above can be rewritten as:

```
+!reach_object(X,Y) | true <<
{
    orientate_towards(X,Y);
    go_to(X,Y);
}</pre>
```

While this "syntactical replacement" has been quite straightforward, dealing with semantics surely implies more problems. The first remark is related to the symbol << which, from semantic point of view, is an *operator* and thus needs two valid *expression* at both left-hand and right-hand side; while for the LHS this is true, it is not the same for the RHS, since it is a *block of code*, not an expression. But even if we could be able to overcome the problem above, and thus find a suitable construct to transform a block of code into an expression, the

```
<sup>5</sup>For instance, we do not want something like "SQL embedded in C".
```

Plan	::=	"(" Head ")" "<<" "(" ActionList ")"
Head		"(" Event ")" " " "(" Condition ")" "(" Event ")" " " "(" Belief ")"
Condition		Belief "&" Belief Condition "&" Belief
Event	::= 	GoalEvent BeliefEvent
GoalEvent	::= 	"+" "!" Belief "-" "!" Belief
BeliefEvent	::= 	"+" Belief "-" Belief
ActionList	::= 	ActionList "," Event ActionList "," AtomicAction

Fig. 3. Example of Operator Grammar for the Host Language

question remains the *meaning* of operators: "+" is the *unaryplus* in C, but it should behave as "*transform the belief or goal into the related event*"; similarly, "!" is the *not* operator, while, in the AgentSpeak view, it should be interpreted as "*achieve the goal represented by the belief*". This "change of meaning" for operators is completely not allowed in C, but if we think to C++ or, more generally, to an object-oriented language, the *operator overloading* feature⁶ can surely help us.

Starting from these considerations, all the actors of a plan, namely *beliefs*, *goals* and *actions*, should become expressions evaluating to *proper objects*, whose class declarations include the proper operator redefinition, suitably allowing the desired "change of meaning". Some *operators* and a *grammar* are needed, in order to guide a designer to properly write the code for operator overloading, which must obviously be in accordance with AgentSpeak syntax. A reference grammar is shown in Figure 3; it can be also used to derive the operators needed to be redefined in the host language, together with their meaning which are summarized in Figure 4.

On the basis of the grammar and operators described, an AgentSpeak plan such as:

```
+object_seen_at(300,400) | true <<
{
    orientate_towards(300,400);
    go_to(300,400);
}</pre>
```

would be rewritten as:

```
class Belief {
   // provide operator overloading here
};
```

```
<sup>6</sup>if supported by the language
```

Operator	Туре	Symbol used	Meaning
		in Figure 3	
Add event	unary	+	Transform a belief or an achievement goal into an addition event
Delete event	unary	-	Transform a belief or an achievement goal into a deleting event
Achieve	unary	!	Transform a belief into an achievement goal
Logical and	binary	&	Concatenate beliefs to represent the condition
Such that	binary		Relate the event with the condition
Implication	binary	<<	Relate the head of the plan (event + condition) with the list of actions
List construct	binary	,	Concatenate actions to represent the body of the plan

Fig. 4. Operators Needed an Their Meaning

```
class Condition {
   // provide operator overloading here
};
```

```
class Action {
   // provide operator overloading here
};
```

```
class object_seen_at : public Belief {
   // provide belief-specific methods
   // and attributes
};
```

```
class true : public Condition {
   // this is be the TRUE condition
};
```

```
class go_to : public Action {
   void execute(void) {
        // the code for the action
   };
};
...
(+object_seen_at(300,400) | true()) <<</pre>
```

According to the code snippet above, when the expression defining the plan is evaluated, first the object_seen_at object is created and the operator unary-"+", redefined in the Belief class, is applied to it; the result (which will surely be another proper object) is then passed, together with the true object, to the code for operator "|"; finally, the result of the last expression will be the first operand for operator "<",

whose RHS is an object of the Action class.

This expression evaluation, which is performed when the corresponding instruction of host language code containing the plan is executed, from the declarative model point of view *does not correspond to plan execution*, but to *plan definition*: indeed, the real actions corresponding to orientate_towards and go_to must be executed when the belief object_seen_at(300,400) is "somewhat asserted".

This separation between *definition* and *execution* of plans has two major consequences. The first one is the need of a proper *processing engine* as a part of the overall runtime system; this engine has to embed structures to represent the *knowledge base* and the *plan library*, also providing an API for their manipulation: the code present in operator redefinition methods will add the objects representing the plan, returned by the plan expression evaluation, into the *plan library*; subsequently, the generation of an external event, such as assertion of a new belief, triggered by means of a suitable API function/method call, will instruct the engine to find the associated plan in the plan library and then really execute the action: this is the reason for the presence of the execute method—for *late execution*—in the Action class.

The second important consequence is the role of *variables* in a plan. If we would like to make the piece of code more general, we should replace constants with variables, that is, something like:

```
(+object_seen_at(X,Y) | true()) <<
    (orientate_towards(X,Y),
      go_to(X,Y) );</pre>
```

However, variables X and Y are interpreted by the host language during plan definition, and thus they not only need to be *defined* but also *bound* to specific values. This is quite undesirable since, according to the declarative paradigm, variable binding must be done during plan execution: like actions need late execution, a *late binding* feature should be provided for variables. To this aim, we cannot use the variables provided by the host languages, which do not permit late binding, and, also in this case, we have to replace them with proper "variable objects". Therefore, by supposing the definition of a class v to represent a variable, the code above becomes:

Proper binding and interpretation of such variables in the context of the execution of a plan is thus a task of the processing engine. This and other implementation aspects will be dealt with in the next Section.

IV. ARCHITECTURE OF PROFETA

We applied the implementation methodology described in the Section above in the design of PROFETA, a framework that allows programmers to define and execute *plans* expressed in an AgentSpeak semantic, but written in a bare Python. This was made possible by the extended operator overloading facilities given by Python itself. In principle, any object– oriented language which provides operator overloading could be used to implement PROFETA, and it is in our plans to translate it also to C++, but Python proved to be very useful to obtain a working proof–of–concept implementation in a couple of weeks.

The main components of the architecture of PROFETA are presented in Figure 5. PROFETA provides the class Attitude and its subclasses Belief and Goal that represent the corresponding concepts of the BDI model (the *mental attitudes* of the Agent). The terms of an attitude are stored in the self._terms attribute. Condition models the condition of a plan, i.e. a set of beliefs separated by '&'. As described in the grammar, the behaviour of this operator is overloaded so that it concatenates the beliefs, storing them in the self.__conditions attribute. Action is just a convenience abstract class: every agent's action must derive Action and implement the execute() method by specifying the instructions to actually perform the action. As for the attitudes, all the parameters of an Action are stored in the self._terms attribute.

An Intention, as defined in the original BDI model, is composed by a list of Actions and does not require a class on itself. Conversely, the class Intentions is a collection of all active Intentions, i.e. of all the plans which have been instantiated and can be executed since their triggering event has happened and the corresponding condition is satisfied.

Notice that the intentions stored into Intentions are a subset of the Plan Library, which contains the set of all plans written as te|cd >> h, according to the grammar previously defined⁷.

On the basis of the proposed methodology, triggering events are defined as follows: the '+' and '-' operator have been over-



Fig. 5. PROFETA class diagram

loaded so that when they precede the instance of an Attitude, an internal field is set accordingly. The corresponding triggering event can be obtained by invoking create_event(). The overloaded '|' operator creates a new Condition object, and store it in the self._condition attribute of Attitude. Finally, the body of the plan is simply a Python list whose elements are actions and/or triggering events. The whole plan is added to the Plan Library using the overloaded ">>" operator.

When plans need variables, those will be bound to actual values on the basis of the content of the knowledge base, and a proper syntax is employed. In PROFETA, the special function $_(..)$ can be used to denote a *variable* within the scope of a plan, e.g. to denote the variable X we will write $_("X")$.

Class Engine represents the *processing engine* which implements the functionalities described in Section III. It holds a reference to the agent's Plan Library and Knowledge Base, implemented in two suitable classes. The KnowledgeBase class exposes an appropriate interface that allows to: (*i*) modify the set of beliefs, (*ii*) test the presence of a particular belief, (*iii*) obtain a specific subset of beliefs.

The basic working scheme of Engine is to run a continuous loop that (i) checks if an event has occurred, (ii) selects an appropriate plan, (iii) evaluates the condition verifying

 $^{^{7}}$ With respect to the grammar in Figure 3, we replaced operator "<<" with ">>" since we argue that it is more appropriate for the concept of "implication".

whether it is true, (iv) executes the plan⁸. While internal events (i.e. events specified in the Body of a plan) are directly handled by Engine, external events that are bound, for example, to perception made by sensors must be notified to the Engine, so that it can reason about them and determine the appropriate plan(s)—if any. In this case, to let the Engine know that an event has occurred, the generate_external_event() method has to be used.

V. CASE STUDY

This Section shows how PROFETA has been actually used to define *strategies* for an autonomous mobile robot taking part in the Eurobot robotic contest [5].

Robots taking part to Eurobot are required to implement a strategy which usually consists in repetitively (i) approach/recognise an object, (ii) pick the object and store it, (iii) put the object in an appropriate place. Objects are usually of different type and colour, and thus need to be sorted, put in different containers or in specific sequences to gain more points, etc. For instance, in the 2010 edition, robots had to collect and store orange and red balls (representing, respectively, oranges and tomatoes) and white cylinders (representing ears of corn). Game matches are played by two robots at the same time, on a shared game table: in order not to get penalties, robots have to avoid each other. Moreover, matches have a 90 seconds time limit. A *winning* game strategy should take into account all the issues stated above and also foresee and handle all the different unexpected situations that could happen during the match.

Figure 6 reports a fragment of an actual game strategy written using PROFETA: during the first part of the match, we wanted our robot to collect ears of corn. According to Eurobot 2010 rules, objects can be found on the game table in fixed, a priori known, positions. We have hardcoded such positions in a global accessible table and enumerated game elementse.g. all ears of corn have been given a code like "c0", "c1", and so on-so that such codes can be used as keys to retrieve the position of the specified object. There exists two kinds of ears of corn: good and fake ones, the formers are painted in white, the latter are painted in black. We wanted-of coursethe robot to collect good ones and to discard the others. On this basis, the employed strategy aims at (i) detecting colour configuration, (ii) picking the first three ears near the starting area (called "c0", "c3" and "c6"), (iii) depositing them into the basket, (iv) picking other three ears near the basket, (v)depositing the objects picked. In picking the ears, we have to pay attention to their colour, and skip it if it is fake (black).

As reported in Figure 6, we defined the beliefs white_corn and black_corn, to represent a ear of corn of the given colour; corns_in_robot, to count the number of ears that the robot has already picked; and no_more_corns to signal that, according to the strategy, we are not interested in picking more ears. Three goals are defined: go, which is the goal triggering the starting of the overall game strategy; grab_corn, aiming at reaching and picking a specified ear of corn; and deposit_corns, which instructs the robot to release the picked ears in the proper basket. Finally, required actions are: detect_configuration, which instructs the vision system to recognise the colour of the various ears and assert the proper white_corn/black_corn beliefs; reach_corn, triggering the robot to move in order to reach the position of a specified ear; pick_corn, driving the mechanical arms to pick the ear from the table (thus properly updating the number of objects inside the robot); reach_deposit_zone, making the robot to reach the basket; and open_tank, which drives the actuators to open the tank and release the objects (this action also resets the number of ears of corn in robot).

As for the strategy, defined in function setup_strategy, the first goal (go) implies to detect the corn colours, grab the first three ears and then go to deposit them. The grab_corn goal is subject to the condition related to ear colour: if it is white, it can be picked, otherwise no action is needed. Goal deposit_corns is instead subject to a condition on the number of ears the robot has picked: if it is less than two⁹, it could be worth to try to pick other ears (and thus save time) before going to the basket. If we have instead an adequate number of ears of corn, we can reach the basket, deposit them, and then go to pick the other ears.

Notice that the strategy example reported in Figure 6 is written in bare Python. Operator overloading does all the magic, making it possible to express all the strategy with a pure declarative syntax. This approach allows to put strategies in a separate Python module or in the same module where Actions are defined, according to programmer's preferences. Moreover, the preferred strategy can be easily selected before each match just by calling the corresponding function, which loads all the rules and initial beliefs into the engine, without the need to recompile of parse again any source file.

VI. CONCLUSIONS AND FUTURE WORK

This paper has described an approach to seamless embed declarative constructs able to write AgentSpeak programs into an imperative object-oriented language. By exploiting operator overloading, a feature proper of many object-oriented languages, and a suitable software architecture, the proposed approach allows a programmer to design, deploy and run a complete agent system, based on a goal-oriented paradigm, using a single programming language and runtime environment. As a proof of concepts, the paper has presented an implementation of the proposed approach in a Python framework, called *PROFETA* (*Python RObotic Framework for dEsigning sTrAtegies*). As it has been described in the case-study, the tool developed has proved its effectiveness in a typical real case, which is the design and implementation of an autonomous mobile robot.

⁸Indeed, the real working scheme of a BDI engine is quite more complex that described; due to space restrictions, we reported only a simplified version; interested readers may refer to [12], [8].

⁹because at least two ears among "c0", "c3" and "c6" are black

```
class white_corn(Belief):
 pass
class black_corn(Belief):
 pass
class corns_in_robot(Belief):
 pass
class no_more_corns(Belief):
 pass
class go(Goal):
 pass
class grab_corn(Goal):
 pass
class deposit_corn(Goal):
 pass
class detect_configuration(Action):
  def execute(self):
   ## ...
class reach_corn(Action):
 def execute(self):
   ## ...
class pick_corn(Action):
  def execute(self):
    ## ...
class reach_deposit_zone(Action):
 def execute(self):
    ## ...
class open_tank(Action):
  def execute(self):
   ## ...
def setup_strategy():
  (+~go()) >> [ detect_configuration(), +~grab_corn("c0"), +~grab_corn("c3"),
                       +~grab_corn("c6"), +~deposit_corns()]
  (+~grab_corn(_("X")) | ( white_corn(_("X")) )) >> [ reach_corn(_("X")), pick_corn() ]
  (+~grab_corn(_("X")) | ( black_corn(_("X")) )) >> [ ]
  (+~deposit_corns() | ( corns_in_robot(_("X")) & (lambda : X > 1)) ) >>
                         [ reach_deposit_zone(), open_tank(), +~deposit_corns() ]
  (+~deposit_corns() | ( corns_in_robot(_("X")) & (lambda : X <= 1) & no_more_corns() )) >>
                         [ # second part of the game ... ]
  (+~deposit_corns() | ( corns_in_robot(_("X")) & (lambda : X <= 1)) ) >>
                         [ +~grab_corn("cl1"), +~grab_corn("cl2"),
                           +~grab_corn("c13"), +no_more_corns(), +~deposit_corns() ]
def start():
 setup_strategy()
  Engine.instance().generate_external_event( +~go() )
```

Fig. 6. An actual game strategy for Eurobot written in Python using PROFETA

Future work will aim at further improving the tool by introducing missing features, such as *test goal* and *goal deletion*, as well as at studying appropriate optimisations to speed up plan selection and execution. Once all the features of PROFETA will be implemented, next step will be the implementation of PROFETA in another object–oriented language, such as C++.

REFERENCES

- F. Bellifemine, A. Poggi, and G. Rimassa, "Jade a fipa-compliant agent framework," in *Proceedings of the Practical Applications of Intelligent Agents*, 1999.
- [2] R. Bordini, A. Bazzan, R. Jannone, D. Basso, R. Vicari, and V. Lesser, "Agent-speak (xl): efficient intention selection in bdi agents via decisiontheoretic task scheduling," in *Proceedings of the 1st International Joint Conference on autonomous agents and multiagent systems*. ACM New York, NY, USA, 2002, pp. 1294–1302.
- [3] J. M. Bradshaw, Ed., Software Agents. AAAI Press/The MIT Press, 1997.
- [4] M. d'Inverno and M. Luck, "Engineering agentspeak(l): A formal computational model," *Journal of Logic and Computation*, vol. 8, no. 3, pp. 233–260, 1998. [Online]. Available: http://eprints.ecs.soton.ac.uk/3846/
- [5] L. Fichera, D. Marletta, V. Nicosia, and C. Santoro, "Flexible robot strategy design using belief-desire-intention model," in *Proceedings of Eurobot Conference 2010*, 2010.
- [6] K. Hinddriks, F. de Boer, W. van der Hoek, and J. C. Meyer, "Agent programming in 3apl," *Int. J. of Autonomous Agents and Multi-agent Systems*, vol. 2, pp. 357–401, 1999.
- [7] —, "Agent programming with declarative goals," in *Intelligent Agents VII. Agent Theories, Architectures and Languages*, ser. LNCS, vol. 1986. Springer–Verlag, 2000.
- [8] J. Hubner, R. Bordini, and M. Wooldridge, "Programming declarative goals using plan patterns," *Lecture Notes in Computer Science*, vol. 4327, p. 123, 2006.
- [9] Jason Home Page, "http://www.jason.sourceforge.net/," 2004.
- [10] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: a multi–agent simulation environment," *Simulation*, vol. 81, no. 7, pp. 517–527, July 2005.
- [11] V. Nicosia, C. Spampinato, and C. Santoro, "Software agents for autonomous robots: The eurobot 2006 experience," in *Proceedings of* the WOA 2006 Workshop, 2006.
- [12] A. Rao, "AgentSpeak (L): BDI agents speak out in a logical computable language," *Lecture Notes in Computer Science*, vol. 1038, pp. 42–55, 1996.
- [13] A. Rao and M. Georgeff, "BDI agents: From theory to practice," in Proceedings of the first international conference on multi-agent systems (ICMAS-95). San Francisco, CA, 1995, pp. 312–319.
- [14] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach/Second Edition. Prentice Hall, 2003.
- [15] C. Santoro, "An erlang framework for autonomous mobile robots," in *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang.* ACM, 2007, pp. 85–92.
- [16] R. Siegwart and I. Nourbakhsh, Introduction to Autonomous Mobile Robots. MIT Press, 2004.
- [17] E. Sklar, "Software review: Netlogo, a multi-agent simulation environment," *Artificial Life*, vol. 13, pp. 303–311, 2007.
- [18] E. Tatara, M. North, T. Howe, N. T. Collier, and J. Vos, "An introduction to repast modelling using a simple predator-prey example," in *Proceedings of Agents 2006 Conference on Social Agents: Results and Prospects*, 2006.
- [19] P. Terna, "Simulation tools for social scientists: Building agent based models with swarm," *Journal of Artificial Societies and Social Simulation*, vol. 1, no. 2, 1998.
- [20] G. Weiss, Ed., Multiagent Systems. The MIT Press, April 1999.
- [21] R. Wray and R. Jones, "An introduction to soar as an agent architecture," in *Cognition and Multi-Agent interaction: from Cognitive Modelling to Social Simulation*, R. Sun, Ed. Cambridge University Press, 2005, pp. 53–78.