Agents' executable specifications

Jacinto A. Dávila[1] and Mayerlin Y. Uzcátegui[12]

[1] Facultad de Ingeniería. CeSiMo
Centro de Simulación y Modelado.
[2] Facultad de Ciencias. SUMA
Sistema Unificado de Microcomputación Aplicada
Universidad de Los Andes.
Mérida 5101, Venezuela.
jacinto,maye@ula.ve

Abstract. This paper presents an specification for an intelligent agent in classical logic. We argue that this type of specification, unlike, for instance, specifications in modal logic, can be systematically translated into an executable code that implements the agent on top of some computing platform. They, therefore, deserve the name of *executable specifications*. We show how to translated our agent's specification into a running implementation and, then, how to migrate that specification from that logical language into an industrial OO specification device (UML) and into an OO programming language (JAVA), in order to build an actual agent.
Keywords: Agents, Logic, Executable Specifications

1 Introduction

This paper presents an specification for an intelligent agent in classical logic. We argue that this type of specification, unlike, for instance, specifications in modal logic, can be systematically translated into an executable code that implements the agent on top of some computing platform. They, therefore, deserve the name of *executable specifications*, as suggested in [21].

In an introductory paper [22], Michael Wooldridge and Nick Jennings distributed the issues associated with the design and construction of intelligent agents in 3 groups: Agent theories, Agent architectures and agent languages. Agent theories provide the answer to the questions: What is an agent? How should it be characterized?. Agent architectures are engineering models of agents: generic blue-prints that could guide the implementation of particular agents (in software or hardware). And Agent languages include all the software engineering tools for programming and experimenting with agents.

This taxonomy of efforts is particularly convenient because it displays the typical development process in agent technologies: first, one decides what the agent (or a multi-agent system) is. Later one decides how to implement it and, then, one decides how to use it to tackle particular problems (if this process fails, the one starts all over again).

In this paper, we present a work that aims to relate the aforementioned categories in a coherent whole. In the proposed framework, an agent theory is

seen as a specification that is systematically converted into an architecture with a built-in agent programming language. Our proposal is to use the same language to state the theory, to implement the architecture and to program the agents. This language is classical logic.

We start by discussing, in sections 2 the uses of logic to specify agents. We explain how an agent theory can be stated in classical logic (rather than, for instance, modal logic as it is usual). In section 3, we show how to translated our agent's specification into a running implementation. In section 4, we show how to migrate that specification from that logical language into an industrial OO specification device (UML) and into an OO programming language (JAVA) in order to build an actual agent. Finally, in section 5, we conclude and define the lines for further research and applications of this tecnology.

2 Agents in Logic

A specification states what a system is and the properties it has. Logic is widely used as a specification language. There are, however, several refinements of logic used as formal languages for specifications.

Agent specifications are normally stated in some version of modal logic [3], [11], [16]. Apparently this is the case because agents are normally regarded as *intentional systems* [7], [15]. An intentional system is one characterized by so-called mental attitudes: beliefs, desires intentions, among others. It is, by definition, an autonomous system, with an internal state upon which it registers inputs and from which it produces outputs. "*Intentional system*" is an useful abstraction in Artificial Intelligence, as it allows the description of entities that have their own *agendas*[15].

It is normally accepted that the description of an agent requires "modalities" to represent intentional notions such as knowledge, beliefs and even goals [20], [11], [22]. Modal logics are very attractive because descriptions in such languages are very close to natural languages. The modeller can designate a modality to indicate knowledge, another modality referring to beliefs and still other to refer to goals. With such a set, the modeller can then state sentences directly translatable into modal logic, such as:

Don Quijote believes Rociñante is a magnificent horse.
Don Quijote wants to be a knight.
Don Quijote believes he knows who Dulcinea's love is.

It is true that modal logic is quite close to natural language and, therefore, specifications are easily translatable into this logic. There is, however, a price to pay. The price is a more complex semantics: possible world semantics. In possible world semantics the fundamental constructs are the possible worlds: conceptual configurations of the universe, each of which represents a state in which the universe could possibly be. Thus, something is possibly true if it is true in some of the worlds. Something is necessarily true if it is true in all possible worlds.

Modern modal logicians [16] found that they could "adapt" possible world semantics to serve other modalities besides possibility and necessity, such as the aforementioned intentional modalities.

We want to show that classical logic can be used, instead of modal logic, to state the specification of agents. Classical logic can be tailored to very realistic descriptions. Also specifications in classical logic has the advantage, we believe, of systematic translation into executable code.

The work presented here started with a proposal by Bob Kowalski which, basically, prescribed the use of logic programs to specify an agent that is both reactive and rational [12], [13]. We developed that proposal into a complete specification by including the specification of a proof procedure [10] that is used as the reasoning mechanism of the agent [4]. This specification is completely translatable into logic programs and is, therefore, an executable specification.

To further illustrate the point, in what follows we show that the executable specification in classical logic without modalitites can be mapped into an UML specification[3] and it can be implement in an object-oriented framework, like the Java platform

Before that, thought, we illustrate how our first-order-logical specification answers questions in agent theory that are answered with modal logics.

2.1 Alleged Deficiencies of Classical Logic to Represent Agents

While justifying their use of modal logic, Jennings and Wooldridge in [22] argued that first-order classical logic (FOL) was not suitable for agent's descriptions. They pointed out two kinds of deficiencies: one in syntax and one in semantics. In the first one, they said that the translation of:

Janine believes Cronos is the father of Zeus

yields a formula that is not first order and, moreover, not even truth-functional, which is:

$$believes(janine, father(zeus, cronos)) \qquad \text{[believes01]}$$

Theirs, we argue, is only one possible way of reading this formula in logic. Jennings and Wooldridge are forced into this non-truth functionality by a wrong translation into first-order classical logic.

Let us say, for instance, that *father/2* is a term in first-order classical logic and a fluent of the situation calculus [14]. Shanahan shows how to state the semantics for these term forms [17] by defining a mapping from the pairs denotating Zeus, as the son, and Cronos, as his father, into *situations*.

[believes01] is, therefore, the proposition that Janine believes in a particular situation (designated by Cronos being the father of Zeus). The formulation

[3] UML: Unified Modelling Language, an object-oriented, visual specification framework.

is still first-order logic and, therefore, truth-functional. Moreover, this truth-functionality does not threaten the intended semantics for *believes/2*, which is to model what an agent believes.

Jennings and Wooldridge's second objection is more serious. It says that the intentional notions (such as beliefs and intentions) are referentially opaque. In our example, for instance, as Zeus and Jupiter "by any reasonable interpretation" [22] denote the same individual, one could write (in first-order logic):

$$zeus = jupiter \qquad \text{[equals01]}$$

and this, they say, would force the agent to deduce:

$$believes(janine, father(jupiter, cronos)) \qquad \text{[believes02]}$$

when it is not the case that Janine believes that Cronos is the father of Jupiter.

This argument, however, confuses the agent that believes in the truth of [equals01] with Janine. It assumes that [equals01] represents common knowledge shared by all the agents. This is a mistake. If Janine believes that Zeus and Jupiter are the same individual, a proper rendering will be something like:

$$believes(janine, equals(zeus, jupiter)) \qquad \text{[believes03]}$$

Only then, one can say that Janine believes that Jupiter is the father of Cronos, provided, of course, that one can associate other elements of "believing" to the agent Janine. One of these elements is the axiom:

$$\forall Ag, X, Y(believes(Ag, father(X, Y)) \leftarrow believes(Ag, father(X, Z)) \wedge believes(Ag, equals(Y, Z))) \qquad \text{[believes04]}$$

or some more general form of the rules that define the beliefs of any agent.

Thus, first-order logic does respect the nature of intentional notions, including this so-called referential opacity. The language of FOL is still useful, on its own, to specify and program agents.

In our example, *believes/2* becomes a description of an agent's basic beliefs and of the rules the agent employs to deduce new beliefs. *believes/2* can also be seen as a predicate in meta-logic. Meta-logic can be used to provide a general solution to agent specification by combining belief processing, as just shown, with goal processing. *believes/2* becomes the demo predicate mentioned below as part of the agent architecture.

3 What is an agent?

An important debate in AI took place when Prof. Rodney Brooks presented a strong case against approaches based on symbolic manipulation and explicit representations [2]. His experiments showed that effective, reactive behaviour was possible without any representation or even a symbol-processing device in the "agent".

Trying to reconcile this reactive behaviour with knowledge representations and with rational, accountable behaviour, Bob Kowalski [12, 13] presented his proposal for an agent that reacts timely to changes and situations in its environment, while it is controlled by a symbolic, rule-based, inference engine.

We completed Kowalski's proposal [4] by producing a detailed specification, in logic, for that reactive and rational agent and a family of languages to program that agent. Our agent has been code-named GLORIA[4].

Fig. 1. GLORIA's specification in logic

$$
\begin{array}{ll}
cycle(KB, Goals, T) & \\
\quad \leftarrow\ demo(KB, Goals, Goals', R) \wedge\ R \leq n & \\
\quad \wedge\ act(KB, Goals', Goals'', T + R) & \\
\quad \wedge\ cycle(KB, Goals'', T + R + 1) & \textbf{[GLOCYC]} \\
act(KB, Goals, Goals', T_a) & \\
\quad \leftarrow\ Goals \equiv PreferredPlan \vee\ AltGoals & \\
\quad \wedge\ executables(PreferredPlan, T_a, TheseActions) & \\
\quad \wedge\ try(TheseActions, T_a, Feedback) & \\
\quad \wedge\ assimilate(Feedback, Goals, Goals') & \textbf{[GLOACT]} \\
executables(Intentions, T_a, NextActs) & \\
\quad \leftarrow\ \forall A, T(do(A, T)\ is\_in\ Intentions & \\
\qquad \wedge\ consistent((T = T_a) \wedge\ Intentions) & \\
\qquad \leftrightarrow\ do(A, T_a)\ is\_in\ NextActs) & \textbf{[GLOEXE]} \\
assimilate(Inputs, InGoals, OutGoals) & \\
\quad \leftarrow\ \forall A, T, T'(action(A, T, succeed)\ is\_in\ Inputs & \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals & \\
\qquad \rightarrow\ do(A, T)\ is\_in\ NGoal) & \\
\quad \wedge\ \forall A, T, T'(action(A, T, fails)\ is\_in\ Inputs & \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals & \\
\qquad \rightarrow\ (\textbf{false} \leftarrow\ do(A, T))\ is\_in\ NGoal) & \\
\quad \wedge\ \forall P, T(obs(P, T)\ is\_in\ Inputs & \\
\qquad \rightarrow\ obs(P, T)\ is\_in\ NGoal) & \\
\quad \wedge\ \forall Atom(Atom\ is\_in\ NGoal & \\
\qquad \rightarrow\ Atom\ is\_in\ Inputs & \\
\quad \wedge\ OutGoals \equiv\ NGoal \wedge\ InGoals & \textbf{[GLOASSI]} \\
A\ is\_in\ B \leftarrow\ B \equiv A \wedge\ Rest & \textbf{[GLOISN]} \\
try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment... & \textbf{[TRY]}
\end{array}
$$

The main component in Kowalski's logical description of an agent is a meta-logic predicate: the cycle predicate, which was extended in the definition of the GLORIA Agent Architecture [4]. Figure 1 shows the main formulae of the specification code-named GLORIA.

In short, the cycle predicate, **[GLOCYC]**, describes a process by which the agent's internal state changes, while the agent assimilates inputs from and posts outputs to the environment (the act predicate, **[GLOACT]** and **[GLOEXE]**), after time-periods devoted to reasoning (the demo predicate, shown below). The things being posted are the *influences*, just as prescribed in the situated multi-agent theory by Ferber and Müller [8] and in our multi-agent simulation theory [5, 6].

[4] GLORIA stands for a **G**eneral-purpose, **L**ogic-based, **O**pen, **R**eactive and **I**ntelligent **A**gent.

Observe the arguments for the demo predicate. This predicate reduces goals to new goals by means of the definitions stored in the knowledge base. "demo" is, precisely, the embodiment of the definition of the "believes" relationship between an agent and its beliefs. An agent believes what she/he/it can DEMOnstrate. This explains the first three arguments of the demo predicate. The fourth argument is an important device to count the amount of resources or the time available for reasoning. At each cycle, the agent reasons for a bounded amount of time and so it is prevented from jumping into infinite regress or total alienation from its environment. This is a legitimate resource in the specification language that allows us, the agent's modellers, to encode an important dimension of the bounded rationality: time for reasoning.

A full description of the demo predicate is in [13], [4]. However, to emphasize the executable condition of the demo predicate, we show, in figures 2 - 3, a simplified version written in PROLOG, which we are using to test our simulation platform [19]. In this version, the agent's knowledge representation is restricted to propositional logic. Observe that goals are represented as a list (i.e. the term goals) of alternative plans. Plans, in turn, are represented as a list (i.e the splan term) of sub-plans. We treat terms that represent actions, sent from the agent to the environment.

Fig. 2. The demo predicate: basic reduction and abductive rules for proposicional logic.

```
% Stop reasoning
demop(_, G, G, [], 0).
demop(_,goals(true,R),R,[],_).
% cut a failing plan
demop(Obs,goals(sp(false,_),Others),G,I,R) :-
   demop(Obs,Others,G,I,R).
% Negation
demop(Obs,goals(sp(not(A),RP),RG),NGoals,Influences,R) :-
   atom(A),NR is R - 1,
   traza(['transform negation into ic ',A,' -> false ']),
   demop(Obs,goals(sp(if(sp(A,true),false),RP),RG),NGoals,Influences,NR).
% Cleaning if false,..  then ..
demop(Obs,goals(sp(if(sp(false,B),C),RP),RG),NGoals,Influences,R) :-
   NR is R - 1,traza(['cleaning if false,',B,' then ',C]),
   demop(Obs,goals(RP,RG),NGoals,Influences,NR).
% Distribution of or within an if
demop(Obs,goals(sp(if(sp(or(A,Rest),B),C),RP),RG),NGoals,Influences,R) :-
   NR is R - 1,
   traza(['distribute ',A,' over ',B,' and ',Rest,' over ',B]),
   agregar_plan(A,B,NewA),demop(Obs,goals(sp(if(NewA,C),
   sp(if(sp(Rest,B),C) ,RP)),RG),NGoals,Influences,NR).
% Propagation.
demop(Obs,goals(sp(if(sp(A,B),C),RP),RG),NGoals,Influences,R) :-
   atom(A),(in(A,RP); member(A,Obs)),NR is R - 1,traza(['propagates ',A]),
   demop(Obs,goals(sp(if(B,C),RP),RG),NGoals,Influences,NR).
% Unfolding within an if
demop(Obs,goals(sp(if(sp(A,B),C),RP),RG),NGoals,Influences,R) :-
   atom(A), unfoldable(A),definition(A,Def), NR is R - 1,
```

```
   traza(['unfold ',A,' into ',Def]),
   demop(Obs,goals(sp(if(sp(Def,B),C),RP),RG),NGoals,Influences,NR).
% Negation in the body of an implication
demop(Obs,goals(sp(if(sp(not(A),B),C),RP),RG),NGoals,Influences,R) :-
   atom(A),NR is R - 1,
   traza(['deals with not ',A,' in the body of an implication' ]),
   demop(Obs,goals(sp(if(B,sp(or(sp(A,true),or(C,false)),true)),RP),RG),
            NGoals,Influences,NR).
% true -> C  is equivalent to C
demop(Obs,goals(sp(if(true,C),RP),RG),NGoals,Influences,R) :- % trace,
   NR is R - 1,traza(['cleans true body of ',C]),
   agregar_plan(C,RP,NP),demop(Obs,goals(NP,RG),NGoals,Influences,NR).
% (A or B) -> C is equivalent to A -> C and B -> C
demop(Obs,goals(sp(if(sp(or(A,B),C),D),RP),RG),NGoals,Influences,R):-
   NR is R - 1,traza(['distributes if ',A,' or ',B ,' then ',C]),
   demop(Obs,goals(sp(if(sp(A,C),D),sp(if((B,C),D),RP)),RG),
        NGoals,Influences,NR).
% (A or B) and C is equivalent to A and C or B and C
demop(Obs,goals(sp(or(A,B),RP),RG),NGoals,Influences,R) :-
   agregar_plan(A,RP,NA),NR is R - 1,
   traza(['distributes ',A,' or ',B,' over ',RP ]),
   demop(Obs,goals(NA,goals(sp(B,RP),RG)),NGoals,Influences,NR).
% Simplification: A and A is equivalent to A
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,R) :-
   atom(A),in(A,RP),NR is R - 1,traza(['simplifies ',A,RP ]),
   demop(Obs,goals(RP,RG),NGoals,Influences,NR).
% Unfolding: A <- Def,A and RP is equivalent to Def and RP
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,R) :-
   atom(A),unfoldable(A),definition(A,Def),NR is R - 1,
   traza(['unfolds ',A,Def]),
   demop(Obs,goals(sp(Def,RP),RG),NGoals,Influences,NR).
%
% Abduction
demop(Obs,goals(sp(A,RP),RG),NGoals,[A| Influences],R) :-
   atom(A),executable(A),traza(['abduces ',A]),NR is R - 1,
   demop(Obs,goals(RP,RG),NGoals,Influences,NR).
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,R) :-
   atom(A),observable(A),member(A,Obs),NR is R - 1,
   traza(['consumes ',A]),
   demop(Obs,goals(RP,RG),NGoals,Influences,NR).
% Removal of plans for not observing on time.
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,R) :-
   atom(A),observable(A),not(member(A,Obs)),NR is R - 1,
   traza(['prunes ',A,RP]),demop(Obs,RG,NGoals,Influences,NR).
% Can't to anything.. but shuffle the first plan..
demop(Obs,goals(sp(A,RP),RG),NG,Influences,R) :-
   agregar_plan(RP,sp(A,true),NP),
   traza(['shuffle',RP,A]),NR is R - 1,
   demop(Obs,goals(NP,RG),NG,Influences,NR).
```

Fig. 3. The demo predicate customized for an example and its invocation.

```
append_plan(true, X, X).
append_plan(splan(A, X), Y, splan(A,Z))
    :- append_plan(X, Y, Z).
in(A, splan(A, _)).
in(A, splan(_,R)) :- in(A, R).
```

```
definition(carry_umbrella,
    or(splan(look_for_it, splan(seize_it, true)),
        splan(borrow_it, true))).
executable(look_for_it).
executable(seize_it).
executable(borrow_it).
observable(it_rains).
% Integrity constraints.
ic(splan(if((it_rains, true), carry_umbrella), true)).
% Recent observations.
obs(splan(it_rains, true)).
%
%   Invoking demo
%
demo(Gin, Gout, Influences) :-
   ic(IC), obs(Obs),
   (Gin = goals(Plan, Rest) ; (Plan = true, Rest = true)),
   append_plan(IC, Plan, NPlan), % IC are put first into the plan.
   append_plan(NPlan, Obs, NNPlan), % Observations are added last.
   demop(goals(NNPlan, Rest),  Gout, Influences, 50).
   % 50 stands .. to bound the time for reasoning
```

4 From the logical specification into an OO implementation

The specification in figure 1 can and has been translated, almost literally, into a Prolog implementation of the agent [4]. We soon understood, however, that the agent could also be implemented in other languages that allow for procedural descriptions. Moreover, that translation from the specification or from its preliminary Prolog implementation into some other procedural language, could be done systematically by exploiting the procedural reading of those logic programs. We set to test this hypothesis step by step.

Firstly, part of that first implementation in Prolog was reduced to the much simpler code in figures 2-3 which corresponds to the agent's inference engine, but for propositional logic only. This is useful for a compact presentation, such as the one in this paper, but also for running the experiments on automatic translation which have been tried in [1].

Before that automatic translations, however, we tried translating the Prolog code, by hand, into Java. Thus, using the cycle predicate as the specification, one could produce this JAVA implementation of an agent:

Fig. 4. An example of the translation into Java.

```
public class Ag {
    List observations;
    List influences;
    Goals goals;
    Beliefs beliefs;
    Goal permanentGoal1;
/** Agent constructor initiates all the structures. */
```

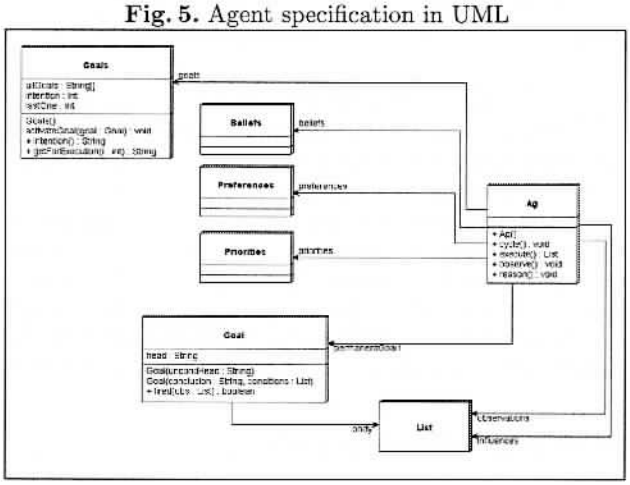
```
    public Ag() {
        observations = null;
        influences = null;
        goals = new Goals();
        beliefs = null;
            // As a test, consider this "permanent goal
        List rains = new List("It rains");
        permanentGoal1 = new Goal("carry umbrella", rains); }
/** The main cycle/locus of control of the agent */
    public void cycle() {
        observe();
        reason();
        List result = execute();
        result.writeAll();
        cycle(); }
/** It's used to try execute the intentions. */
    public List execute() { return influences; }
/** It's used to update the knowledge of the environment. */
    public void observe() {
            // Get inputs from the environment.. somehow..
        List obs = new List("it rains");
            // Update its records.
        observations = obs; }
/** Very simple implementation of the reasoning engine. */
    public void reason() {
            // Every goal must be checked against observations..
        if (permanentGoal1.fired(observations)) {
            goals.activateGoal(permanentGoal1); };
        influences = new List(goals.allGoals[goals.intention]); }
/** Auxiliary method to test the agent. */
    public static void main(String argv[]) {
        Ag agent = new Ag();
        agent.permanentGoal1.body.writeAll();
        agent.cycle(); }}
```

From this exercise we also obtained, by reverse engineering, the general UML specification for an agent shown in figure 5.

In this OO framework, the class `Ag` contains the methods that implement the agent which roughly correspond to the predicates in figure 1. It does include the methods that correspond to the cycle predicate (`cycle()`), the act predicate (`observe()` and `execute()`) and the demo predicate (`reason()`), all of them adjusted to deal with simpler representations of goals and beliefs in the agent. Beliefs, in fact, are not represented at all in this first translation into Java. The method `execute()` communicates the agent's intention to the environment. `observe()`, on the contrary, communicates a description of the environment to the agent. `reason()` implements the reasoning engine that mediates between perceptions and actions. The accompanying classes implements the data structures (and associated methods) required to store and manage agent's goals and beliefs, among other things. Note that a basic data structure: a List class, is an essential part of the implementation. We require a list with accessible head and tail components only. For the sake of space, in figure 5 only the `Goals`' attributes and methods are shown.

The simplification of goals is important as much of the processing in the agent's inference engine depends upon those data structures. We provided, by

Fig. 5. Agent specification in UML



hand, the basic data structures to store goals as terms, representing condition-action rules. The basic Java class for these purposes in the List class, which supports the representation of conjunction of conditions in the body of a condition-action rule. The head of these rules is always a low-level, executable action for the agent. This is a critical restriction with respect to the original specification which allows for complex, compound actions, that can be reduced, with further reasoning, to executable, atomic actions. For this, the more elaborated engine described in figures 2-3 is required.

Observe that in the Java implementation, and also for the sake of simplicity, a plan contains only one action. The `Goals` class implements the set of alternative plans for the agent.

In the example in figure 4, we test the agent with a very simple rule: "if it rains then carry an umbrella", which is codified as a list of strings. Observe that the inputs corresponds to the string "it rains", again for simplicity.

It is worth recalling that the core of the JAVA code is obtained as a direct translation from the logic formulae. This has encouraged us to start exploring automatic, partial translation procedures. In [1], this is nicely achieved by a set of Java programs that transform Prolog code into a collection of Java classes, a class for each predicate definition in the original code. As a very brief account of the strategy, let us imagine that the agent original code in Prolog includes the clause:

```
agent(In,Out) :- reason(In,InTG), act(InTG, Out).
```

In this case, the programs written by Amaya [1] will produce a Java class `Agent_2` whose internal structure is something like this:

```
...
Reason_2 reason_2i = new Reason_2();
Act_2 act_2i = new Act_2(); ...
while(i1=0 or i1< reason_2i.getNumberTerms())  { ...
```

```
if(reason_2i.searchT(term1, term2, int)=true){
    if(act_2i.searchT(term2, term3, int)=true){ ...
        back = true; ...
        return back; }}
i1++; } ...
```

where the object instances `reason_2i` and `act_2i` represent the predicates that are being invoked in the body of *agent/2*, and the while and if structures implement the search tree associated to this predicate. Amaya has gone on to allow for search strategies specific to each predicate.

We want to gather as much experience as possible in systematic or automatic translations because we expect inference engine customization to be part of the regular practice in agent development. With a systematic or semi-automatic methodology, we could concentrate on customization at the level of the specification, where concepts and trade-off are clearer, and then produce the executable code with minimal additional effort and, more importantly, minimal error.

5 Conclusions and further work

This paper describes the formal specification of an agent. The language used to state the specification is a form of classical logic, as opposed to modal logic. We have shown that this specification can be systematically translated into other formalisms and, more importantly, into executable code. It is, therefore, a mechanism for the "agentfication process" described by Shoham [18] by means of which a specification produces an agent. We believe this agentfication process can greatly benefit from using *executable specifications*, where specification/implementation trade-offs can be more easily understood.

This work is part of a project to build a multi-agent simulation platform called GALATEA. We previously presented the specification of the simulation platform [5] and described the multi-agent and OO simulation platform [6]. The following steps are 1) to assembly the platform with the interface between agents and the simulation engine 2) to develop alternative inference engines for the agent (a different embodiment of the demo predicate) and 3) to perform the first multi-agent simulation experiments.

Acknowledgements

This work has been partially funded by CDCHT-University of Los Andes projects I-524-95-02-AA, I-666-99-02-E and I-667-99-02-B.

References

1. Jhon E. Amaya, *Integración de la programación declarativa y la programación orientada por objetos para desarrollar agentes inteligentes*, Master's thesis, Universidad de Los Andes, Mérida, Venezuela, June 2003.

2. Rodney Brooks, *Intelligence without representation*, Artificial Intelligence (1991), 139–159.
3. P. R. Cohen and H. J. Levesque, *Intention is choice with commitment*, Artificial Intelligence **42** (1990), 213–261.
4. Jacinto A. Dávila, *Agents in logic programming*, Ph.D. thesis, Imperial College of Science, Technology and Medicine, London, UK, June 1997.
5. Jacinto A. Dávila and Kay A. Tucci, *Towards a logic-based, multi-agent simulation theory*, International Conference on Modelling, Simulation and Neural Networks [MSNN-2000] (Mérida, Venezuela), AMSE & ULA, October, 22-24 2000, pp. 199–215.
6. Jacinto A. Dávila and Mayerlin Uzcátegui, *Galatea: A multi-agent simulation platform*, International Conference on Modelling, Simulation and Neural Networks [MSNN-2000] (Mérida, Venezuela), AMSE & ULA, October, 22-24 2000, pp. 217–233.
7. Daniel Dennett, *The intentional stance*, The MIT Press, Cambridge, MA, 1987.
8. Jacques Ferber and Jean-Pierre Müller, *Influences and reaction: a model of situated multiagent systems*, ICMAS-96, 1996, pp. 72–79.
9. Martin Fowler and Kendall Scott, *Uml distilled: A brief guide to the standard object modeling language*, second ed., Addison-Wesley Pub Co, 1999.
10. T. H. Fung and Robert A. Kowalski, *The iff proof procedure for abductive logic programming*, Journal of Logic Programming (1997).
11. Nicholas R. Jennings, *Controlling cooperative problem solving in industrial multi-agent systems using joint intentions*, Artificial Intelligence **74** (1995), no. 2.
12. Robert A. Kowalski, *Using metalogic to reconcile reactive with rational agents*, Meta-Logics and Logic Programming (K. Apt and F. Turini, eds.), MIT Press, 1995.
13. Robert A. Kowalski and Fariba Sadri, *Towards a unified agent architecture that combine rationality with reactivity*, LID'96 Workshop on Logic in Databases (San Miniato, Italy) (Dino Pedreschi and Carlos Zaniolo, eds.), July 1996.
14. J. McCarthy and P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence **4** (1969), 463–502.
15. John. McCarthy, *Making robots conscious of their mental states*, Machine Intelligence **15** (1995).
16. Moore, *Logic and representation*, Center for the Study of Language and Information (CSLI), 333 Ravenswood Avenue, Menlo Park, CA 94025, 1995, ISBN 1-881526-15-1.
17. Murray Shanahan, *Solving the frame problem: A mathematical investigation of the common sense law of inertia*, MIT Press, 1997.
18. Yoav Shoham, *Agent-oriented programming*, Technical Report STAN-CS-1335-90, Stanford University, Stanford, CA 94305, 1990.
19. Mayerlin Y. Uzcátegui, *Diseño de la plataforma de simulación de sistemas multi-agentes galatea*, Master's thesis, Maestría en Computación, Universidad de Los Andes. Mérida. Venezuela, 2002, Tutor: Dávila, Jacinto.
20. Michael Wooldridge, *The logical modelling of computational multi-agent systems*, Ph.D. thesis, Department of Computation, Manchester Metropolitan University, Manchester, UK, October 1992.
21. Michael Wooldridge and P Ciancarini, *Agent-oriented software engineering: The state of the art*, Springer-Verlag Lecture Notes in AI. **1957** (2001).
22. Michael Wooldridge and Nicholas R. Jennings, *Intelligent agents: Theory and practice*, Knowledge Engineering Review **10** (1995), no. 2, 115–152.