Mayerlin Uzcátegui

From: "Jacinto A. Dávila" <jacinto@ula.ve>
To: "Mayerlin Y. Uzcategui S." <maye@ula.ve>
Sent: viernes, 04 de octubre de 2002 9:55
Subject: LPAR 2002 notification (fwd)

--
Jacinto A. Dávila (http://cesimo.ing.ula.ve/~jacinto/)
Coordinador del Postgrado en Modelado y Simulación (cesimo.ing.ula.ve)
Coordinador del Laboratorio de Computación Avanzada, Multimedia y
Video-Conferencia (pgvirtual.pg.ing.ula.ve/labcamv/).
Universidad de Los Andes (www.ula.ve)

---------- Forwarded message ----------
Date: Mon, 22 Jul 2002 10:39:16 +0100
From: LPAR 2002 <lpar2002@cs.man.ac.uk>
Reply-To: voronkov@cs.man.ac.uk, baaz@logic.at
To: jacinto@ula.ve
Subject: LPAR 2002 notification

Paper #: 65
Title: Agents' executable specifications.

Dear authors,

we are sorry to inform you that you paper has not been
been accepted to LPAR 2002. We received 68 papers and only 30
of them have been accepted. You will receive reviews
on your paper shortly.

However, we decided to invite some of the rejected papers, including
your paper, for a presentation at a poster session. All papers
presented at the poster session will be published as a volume
of the Proceedings of the Kurt Goedel Society after the conference.
Maaathias Baaz will contact you about the details of how to prepare
yor paper for the poster session proceedings. Please let us know if
you are interested to participate in the poster session.

Best regards,
Matthias Baaz, Andrei Voronkov (LPAR 2002 programme chairs).

GLORIA: An Agent's Executable Specification

Jacinto Dávila[1] and Mayerlin Uzcátegui[1 2]

[1] CESIMO,
[2] SUMA,
Universidad de Los Andes.
Mérida, 5101. Venezuela
{jacinto,maye}@ula.ve

Abstract. This paper presents a specification for an intelligent agent in classical logic. We argue that this type of specification 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 [17]. We illustrate 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.

1 Introduction

In an introductory paper [18], 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 [17].

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. The language is classical logic.

2 Agents in Logic

A specification states what a system is and the properties it has. Logic is widely used as a specification language. Agent specifications are normally stated in some version of modal logic [2], [9], [13]. Apparently this is the case because agents are normally regarded as *intentional systems* [6], [12]. An intentional system is one

characterized by so-called 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*[12]. It is widely believed that the representation of autonomous systems requires the alleged added expressiveness of the modal logics, with their possible world semantics. Typically, a number of new modal operators (such as K for knowledge, B for beliefs and G for goals) are introduced. In this work, we depart from that trend and refrain from using modal logics. Instead, we use classical logic and employ a set of meta-logic predicates to model, not the agents beliefs and goals separately, but some processes associated with them.

2.1 The proposal by Kowalski

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 [10], [11]. We developed that proposal into a complete specification by including the specification of a proof procedure [8] that is used as the reasoning mechanism of the agent [3]. This specification is completely translatable into logic programs and is, therefore, an executable specification. Our agent has been code-named GLORIA[1].

2.2 Our version of the cycle predicate

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 (with 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 [7] and in our multi-agent simulation theory [4,5].

2.3 The demo predicate: an abductive reasoner

Observe the arguments for the demo predicate. This predicate reduces goals to new goals by means of 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 for jumping into infinite regress or total alienation

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

Fig. 1. GLORIA's specification in logic

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

GLORIA's cycle

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 found in realistic agents, as we have been arguing.

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

```
% A is allowed if it is not a reserved word or term structure.
allowed(A):-  A \= true,A \= false,A \= not(_),A \= if(_,_),
  A \= sp(_,_),A \= or(_,_),A \= goals(_,_).
% Stop reasoning when a plan is completed.
demop(_,goals(true,R),R,I,I,_).
% cut a failing plan
demop(Obs,goals(sp(false,_),Others),G,_,IOut,R):-
  demop(Obs,Others,G,[],IOut,R).
% Stop reasoning when there are no more resources
demop(_,G,G,I,I,0).
% Equality
```

```
demop(Obs,goals(sp(X=Y,RP),RG),NGoals,Influences,InfOut,R):-
  apply(X,Y,RP,NewRP),NR is R - 1,,
  demop(Obs,goals(NewRP,RG),NGoals,Influences,InfOut,NR).
% Negation
demop(Obs,goals(sp(not(A),RP),RG),NGoals,Influences,InfOut,R ):-
  allowed(A),NR is R-1,,demop(Obs,goals(sp(if(sp(A,true),false),
  RP),RG),NGoals,Influences,InfOut,NR).
% Cleaning if false,..  then ..
demop(Obs,goals(sp(if(sp(false,B),C),RP),RG),NGoals,Influences,
  InfOut,R):- NR is R - 1,,
  demop(Obs,goals(RP,RG),NGoals,Influences,InfOut,NR).
% Distribution of or within an if
demop(Obs,goals(sp(if(sp(or(A,Rest),B),C),RP),RG),NGoals,
  Influences,InfOut,R):-NR is R - 1,,
  agregar_plan(A,B,NewA),demop(Obs,goals(sp(if(NewA,C),
  sp(if(sp(Rest,B),C),RP)),RG),NGoals,Influences,InfOut,NR).
% Propagation.
demop(Obs,goals(sp(if(sp(A,B),C),RP),RG),NGoals,Influences,InfOut,R):-
  allowed(A),(in(A,RP); member(A,Obs)),NR is R - 1,,
  demop(Obs,goals(sp(if(B,C),RP),RG),NGoals,Influences,InfOut,NR).
% Equality within an if (notice we are not considering splitting as yet)
demop(Obs,goals(sp(if(sp(X=Y,B),C),RP),RG),NGoals,Influences,InfOut,R):-
   apply(X,Y,B,NewB),apply(X,Y,C,NewC),apply(X,Y,RP,NewRP),NR is R-1,,
   demop(Obs,goals(sp(if(NewB,NewC),NewRP),RG),NGoals,Influences,InfOut,NR).
% Unfolding within an if
demop(Obs,goals(sp(if(sp(A,B),C),RP),RG),NGoals,Influences,InfOut,R):-
   allowed(A),unfoldable(A),definition(A,Def),NR is R - 1,,demop(Obs,
   goals(sp(if(sp(Def,B),C),RP),RG),NGoals,Influences,InfOut,NR).
% Negation in the body of an implication
demop(Obs,goals(sp(if(sp(not(A),B),C),RP),RG),NGoals,Influences,
  InfOut,R):- allowed(A),NR is R-1,,demop(Obs,goals(sp(if(B,sp(or(sp(A,
  true),or(C,false)),true)),RP),RG),NGoals,Influences,InfOut,NR).
% true -> C  is equivalent to C
demop(Obs,goals(sp(if(true,C),RP),RG),NGoals,Influences,InfOut,R):-
  NR is R - 1,,
  demop(Obs,goals(NP,RG),NGoals,Influences,InfOut,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,
  InfOut,R):- NR is R - 1,,demop(Obs,goals(sp(if(sp(A,C),D),
  sp(if((B,C),D),RP)),RG),NGoals,Influences,InfOut,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,InfOut,R):-
  agregar_plan(A,RP,NA),,
  demop(Obs,goals(NA,goals(sp(B,RP),RG)),NGoals,Influences,InfOut,NR).
% Simplification: A and A is equivalent to A unification permitting
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,InfOut,R):-
  allowed(A),in(A,RP),% unification will be perfomed
  NR is R - 1,,
  demop(Obs,goals(RP,RG),NGoals,Influences,InfOut,NR).
% Unfolding: A <- Def,A and RP is equivalent to Def and RP
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,InfOut,R):-
  allowed(A),unfoldable(A),definition(A,Def),NR is R - 1,,
  demop(Obs,goals(sp(Def,RP),RG),NGoals,Influences,InfOut,NR).
% Abduction to produce influences..
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,InfOut,R):-
  allowed(A),executable(A),,
  demop(Obs,goals(RP,RG),NGoals,NInfluences,InfOut,NR).
% Abduction to consume observations. Experimental.
demop(Obs,goals(sp(A,RP),RG),NGoals,Influences,InfOut,R):-
  allowed(A),observable(A),member(A,Obs),NR is R - 1,,
```

```
  demop(Obs,goals(RP,RG),NGoals,Influences,InfOut,NR).
% Removal of plans for not observing on time.
demop(Obs,goals(sp(A,RP),RG),NGoals,_,InfOut,R):-
  allowed(A),observable(A),not(member(A,Obs)),NR is R - 1,,
  demop(Obs,RG,NGoals,[],InfOut,NR).
% Can't to anything.. but shuffling the first plan..
demop(Obs,goals(sp(A,RP),RG),NG,Influences,InfOut,R):-
  agregar_plan(RP,sp(A,true),NP),,NR is R - 1,
  demop(Obs,goals(NP,RG),NG,Influences,InfOut,NR).
% Add plans with the structure sp(First Action,Rest of Plan)
agregar_plan(true,X,X).
agregar_plan(sp(A,X),Y,sp(A,Z)):-agregar_plan(X,Y,Z).
%
in(A,sp(A,_)).
in(A,sp(_,R)):- in(A,R).
```

2.4 An example

A full description of the demo predicate is in [11], [3]. However, to emphasize the executable condition of the demo predicate, we show, in figures 2 and 3, a simplified version written in PROLOG, which we are using to test our simulation platform [16]. 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 term splan) of sub-plans. We treat terms that represent actions, sent from the agent to the environment, and terms that represent observations, sent from the environment into the agent, as abducibles, in the sense explained in [8,3].

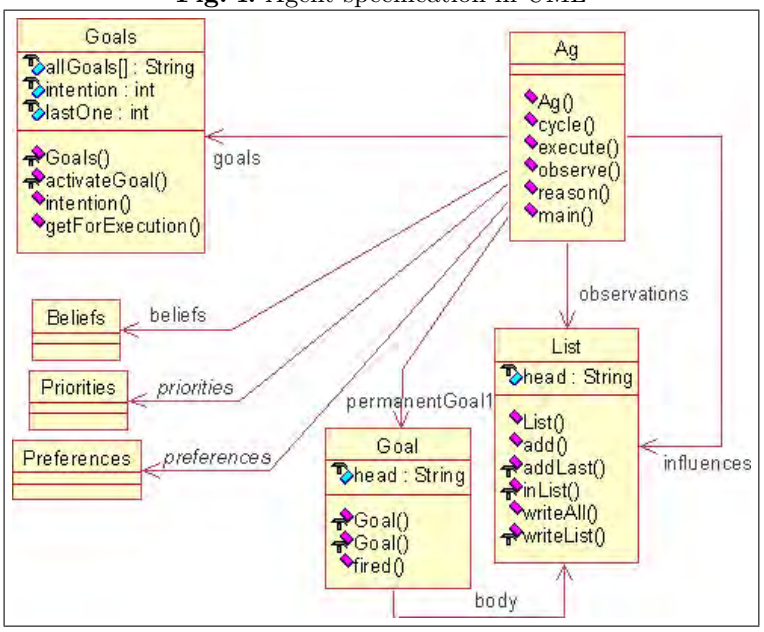
Fig. 3. The demo predicate customized for an example and its invocation.

```
definition(G,Plan):-
  createscheme(G,NG),findall((NG,B),(to NG do B),L),
  attach_eq(G,L,D),make_or(D,Plan).
% Transforma  [] list into a(..,false) or-list
make_or([],false). make_or([A|B],or(AA,BB)):-
arregla(A,AA),make_or(B,BB).
%
attach_eq(_,[],[]):- !.
attach_eq(G,[(NG,C)|RestC],[NC|NRestC] ):-unpack_eq_falsefilter(G,NG,Eq),
  !,and_append(Eq,C,NC),attach_eq(G,RestC,NRestC).
attach_eq(G,[_|RestC],NRestC):-  attach_eq(G,RestC,NRestC).
%
unpack_eq_falsefilter(G,Gs,T_eq_S):-  unpack_eq(G,Gs,T_eq_S).
%
createscheme(Predicate,PredicateScheme):- Predicate =..
  [Predicatename|Arguments],duplicate_struct(Arguments,NewArg),
  PredicateScheme =.. [Predicatename|NewArg].
%
duplicate_struct([],[]). duplicate_struct([_|R],[_|CR]):-
  duplicate_struct(R,CR).
%
```

```
unpack_eq(Pt,Ps,T_eq_S):-  Pt =.. [P|T], Ps =.. [P|S],
  build_eq(T,S,T_eq_S),!.
unpack_all_eq(_,[],[],[]).
unpack_all_eq(G,[Gs|Rest],[T_eq_S|D],FinalAll):-
  unpack_eq(G,Gs,T_eq_S),(T_eq_S =([],true) -> FinalAll = RestA;
  FinalAll = [Gs|RestA] ),unpack_all_eq(G,Rest,D,RestA).
%
build_eq([],[],true).
build_eq([T1|RT],[S1|RS],(T1 = S1,Rest)):- build_eq(RT,RS,Rest).
%
and_append(F,S,R):- F == true,and_compress(S,R).
and_append(F,S,(F,R)):-
 (var(F);(functor(F,Funct,_),Funct \==',')),!,and_compress(S,R).
and_append((F,Rf),S,(F,Ro)):- F \== true,!, and_append(Rf,S,Ro).
and_append((F,Rf),S, Ro):- F == true,and_append(Rf,S,Ro ).
%
and_compress(L,L):- L == true,!.
and_compress(L,(L,true)):-(var(L);(functor(L,Funct,_),Funct \==',')),!.
and_compress((F,Rf),(F,Ro)):- F \== true,!,and_compress(Rf,Ro).
and_compress((F,Rf), Ro):- F == true,!,and_compress(Rf,Ro).
%
ic(Restricciones):-
  findall(if(BBody,HHead),((if Body) then Head),arregla(Body,BBody),
  arregla(Head,HHead)),L), aplana(L,Restricciones).
%
obs(L):- findall(Obs,observe Obs,L).
%
aplana([],true). aplana([C|R],sp(C,RR)):- aplana(R,RR).
%
arregla(true,true):- !. arregla((A,B),sp(A,BB)):- !,arregla(B,BB).
arregla(A,sp(A,true)).
%
% Invoking demo
demo(Gin,Gout,Influences):-  ic(IC),obs(Obs),(Gin = goals(Plan,Rest);
 (Plan = true,Rest = true)),agregar_plan(IC,Plan,NPlan),
  demop(Obs,goals(NPlan,Rest), Gout,[],Influences,300).
demod(Obs,Gin,Gout,Influences):-  ic(IC),(Gin = goals(Plan,Rest);
  (Plan = true,Rest = true)),agregar_plan(IC,Plan,NPlan),
  demop(Obs,goals(NPlan,Rest), Gout,[],Influences,200).
%
unfoldable(A):- \+ executable(A),\+ observable(A).
%
apply(X,Y,C,NC):- var(X),nonvar(Y),apply_conj([X/Y],C,NC),!.
apply(X,Y,C,NC):- var(Y),nonvar(X),apply_conj([Y/X],C,NC),!.
apply(X,X,C,C).
apply_conj([],C,C):- !. apply_conj(_,true,true).
apply_conj([V/T|Rest],Conjunct, NewConj):- substitute_conj(V,T,
  Conjunct,NextConj), apply_conj(Rest,NextConj,NewConj).
%
substitute_conj(_,_,true,true):- !.
substitute_conj(V,T,(G,Rest),(NewG,NewRest)):-
  substitute(V,T,G,NewG),substitute_conj(V,T,Rest,NewRest).
substitute(V,T,Pred,NewPred):- Pred =.. [Name|Args],
  substitute_args(V,T,Args,NewArgs),NewPred =.. [Name|NewArgs].
substitute_args(_,_,[],[]).
substitute_args(V,T,[A|Rest],[T|NRest]):-
  V == A,!,substitute_args(V,T,Rest,NRest).
substitute_args(V,T,[A|Rest],[NewA|NRest]):- % complex atoms.
  compound(A),!,substitute(V,T,A,NewA),substitute_args(V,T,Rest,NRest).
substitute_args(V,T,[A|Rest],[A|NRest]):-
  V \== A,substitute_args(V,T,Rest,NRest).
```

```
% ------------------------------------------------------
executable(get_umbrella(_)).
executable(go_home(_)).
observable(it_rains(_)).
%
if it_rains(X) then protect_yourself(X).
%
to protect_yourself(X) do get_umbrella(X).
to protect_yourself(X) do go_home(X).
```

Fig. 4. Agent specification in UML



3 From logic to UML

The cycle predicate and the structures it uses (mainly to store the knowledge base and the goals) lead to a general Unified Modelling Language specification for an agent similar to the one shown in figure 4.

In this OO framework, the class `Ag` contains the methods that implement the agent. It includes the methods that correspond to the cycle predicate (`cycle()`), the act predicate (`observe()` and `execute()`) and the demo predicate (`reason()`).

3.1 From logic to 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, is an essential part of the implementation. We only require a list with accessible head and tail components. For the sake of space, in figure 4 only the `Goals`' attributes and methods are shown.

Thus, using the cycle predicate as the specification, one can produce a JAVA implementation of an agent like the one in figure 5. For the sake of simplicity, in this implementation a plan contains only one action. The `Goals` class implements the set of alternative plans for the agent.

Fig. 5. Ag Class

```
/** This is a preliminary implementation
 *  of the specification in GLORIA.
 */
/*** Ag is the class that implement the agent */
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");
       /* it corresponds to "if it rains then carry an umbrella" */
    permanentGoal1 = new Goal("carry umbrella", rains);}
    /*** It is used to try execute the intentions. */
        public List execute() {
            return influences; }
    /*** It is used to update the knowledge of the environment. */
        public void observe() {
            // Get inputs from the environment..
            // its being simulated with a single report of "it rains"
            List obs = new List("it rains");
            // Update its records.
            observations = obs; }
    /*** This is a 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]); }
    /*** The main cycle/locus of control of the agent */
        public void cycle() {
            observe();
```

```
            reason();
            List result = execute();
            result.writeAll();
            cycle(); }
  /*** This is an auxiliary method to test the agent. */
    public static void main(String argv[]) {
        Ag agent = new Ag();
        agent.permanentGoal1.body.writeAll();
        agent.cycle(); }
} // End of Ag class
```

4 Conclusions

This paper presented 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 [15] 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 has been applied to the construction of a multi-agent simulation platform called GALATEA. We previously presented the specification of the simulation platform [4] and described the multi-agent and 00 simulation platform [5]. 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. [1], [14]

Acknowledgements

This work has been partially funded by CDCHT-University of Los Andes projects I-666-99-02-E and I-667-99-02-B and FONACIT project S1-2000000819.

References

1. M. Ablan, J. Dávila, N. Moreno, R. Quintero, and M. Uzcátegui, *Agent modeling of the caparo forest reserve*, EUROSIS 2003 (Napoles, Italy), 2003.
2. P. R. Cohen and H. J. Levesque, *Intention is choice with commitment*, Artificial Intelligence **42** (1990), 213–261.
3. Jacinto A. Dávila, *Agents in logic programming*, Ph.D. thesis, Imperial College of Science, Technology and Medicine, London, UK, June 1997.
4. 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.

5. 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.
6. Daniel Dennett, *The intentional stance*, The MIT Press, Cambridge, MA, 1987.
7. Jacques Ferber and Jean-Pierre Müller, *Influences and reaction: a model of situated multiagent systems*, ICMAS-96, 1996, pp. 72–79.
8. T. H. Fung and Robert A. Kowalski, *The iff proof procedure for abductive logic programming*, Journal of Logic Programming (1997).
9. Nicholas R. Jennings, *Controlling cooperative problem solving in industrial multi-agent systems using joint intentions*, Artificial Intelligence **74** (1995), no. 2.
10. 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.
11. 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.
12. John. McCarthy, *Making robots conscious of their mental states*, Machine Intelligence **15** (1995).
13. 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.
14. R. Quintero, R. Barros, J. Dávila, N. Moreno, Tonella G., and M. Ablan, *A model of the biocomplexity of deforestation in tropical forest: Caparo case study*, iEMSs 2004 (Osnabrueck, Germany), 2004.
15. Yoav Shoham, *Agent-oriented programming*, Technical Report STAN-CS-1335-90, Stanford University, Stanford, CA 94305, 1990.
16. 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.
17. Michael Wooldridge and P Ciancarini, *Agent-oriented software engineering: The state of the art*, Springer-Verlag Lecture Notes in AI. **1957** (2001).
18. Michael Wooldridge and Nicholas R. Jennings, *Intelligent agents: Theory and practice*, Knowledge Engineering Review **10** (1995), no. 2, 115–152.