Logic-Based Languages to Model and Program Intelligent Agents

Maurizio Martelli[1], Viviana Mascardi[1], and Leon Sterling[2]

[1] DISI, Università di Genova,
Via Dodecaneso 35, 16146, Genova, Italy.
{martelli,mascardi}@disi.unige.it
[2] Department of Computer Science and Software Engineering,
The University of Melbourne Victoria 3010, Australia.
leon@cs.mu.oz.au

Abstract Research on tools for modeling and specifying intelligent agents, namely computer systems situated in some environment and capable of flexible autonomous actions, is very lively. Due to the complexity of intelligent agents, the way they are modeled, specified and verified should greatly benefit by the adoption of formal methods. Logic-based languages can be a useful tool for engineering the development of a multi-agent system (MAS). This paper discusses six logic-based languages which have been used to model and specify agents, namely ConGolog, AGENT-0, the IMPACT specification language, Dylog, Concurrent METATEM and $\mathcal{E}_{hhf}$. To show their main features and to practically exemplify how they can be used, a common running example is provided. Besides this, a set of desirable features that languages should exhibit to prove useful in engineering a MAS have been identified. A comparison of the six languages with respect to the support given to these features is provided, as well as final considerations on the usefulness of logic-based languages for "agent oriented software engineering".

1 Introduction

Research on formal methods for modeling and programming real-world applications characterized by *autonomous*, *distributed* and *heterogeneous* components which *sense* and *react* to changes occurring in their environment and are able of *complex* interaction is a very active research area which spans across the fields of artificial intelligence and software engineering. The metaphor underlying this research is that any component of the application is an *intelligent agent*, namely, according to a classical definition [10],

> "… a computer system, *situated* in some environment, that is capable of *flexible autonomous* actions in order to meet its design objectives."

As pointed out in [2] the way intelligent agents are modeled, specified and verified should greatly benefit by the adoption of *formal methods*. Logic is recognized to be a powerful means for specifying agents, and when the logic is executable it becomes a powerful tool to rapidly develop prototypes of the agents and to verify their properties.

This paper provides an overview of six logic-based languages which have proven useful to specify and prototype intelligent agents: ConGolog (Section 3), AGENT-0 (Section 4), the IMPACT specification language (Section 5), Dylog (Section 6), Concurrent METATEM (Section 7) and $\mathcal{E}_{hhf}$ (Section 8). The main strength of these languages lies in their suitability to easily and quickly specify agents and MASs and to fast develop working prototypes (each language can be executed thanks to a working interpreter). In our opinion, however, most of these languages are not very suitable to implement final real applications; this paper will then concentrate on the features these languages provide for modeling, specifying and prototyping, and will not deal with them as implementation languages. For each language we take under consideration, we provide a short description and we practically exemplify the way the language can be used by means of the running example depicted in Section 2. In Section 9 we identify a set of features that, in our opinion, languages for agent-oriented software engineering should provide: explicit representation of time, ability to sense the environment, support given to communication, concurrency and nondeterminism, modularity and clear semantics. We also draw a comparison of the languages along these dimensions . In Section 10 we conclude our work with some considerations on advantages of the use of executable logic-based languages for modeling and programming agents and with our future directions of work. An extended version of this paper can be found in Chapter 3 of [11].

2 The running example

To show how to define an agent in the various agent languages we discuss in this paper, we implement a simple seller agent working in a distributed marketplace in each language. The seller agent may receive a contractProposal message from a buyer agent. According to the amount of goods required and the price proposed by the buyer, the seller may accept the proposal and send the goods, refuse it or try to negotiate a new price by sending a contractProposal message back to the buyer.

For sake of synthesis, we only deal with the first situation: if the received message is contractProposal(goods, amount, proposed-price) and there is enough goods in the warehouse and the price is greater or equal than a max value, then the seller accepts the proposal by sending an accept message to the buyer and concurrently ships the required goods to the buyer[1]. In our example, the goods to be exchanged are oranges, with minimum and maximum price 1 and 2 euros respectively. The initial amount of oranges that the seller possesses is 1000.

3 ConGolog

ConGolog [8] is a concurrent programming language based on the situation calculus (J. McCarthy [13]) which includes facilities for prioritizing the concurrent execution, interrupting the execution when certain conditions become true, and dealing with exogenous actions. ConGolog includes the following constructs:

[1] If it is not possible to define concurrent actions because the language does not support them, answering and shipping goods will be executed sequentially.

a	Primitive action.
$\Phi?$	Wait for a condition.
$(\delta_1; \delta_2)$	Sequence.
$(\delta_1 \mid \delta_2)$	Nondeterministic choice between actions.
$\pi v.\delta$	Nondeterministic choice of arguments.
δ^*	Nondeterministic iteration.
$\{\textbf{proc}\ P_1(\vec{v_1})\delta_1\ \textbf{end}; \ldots \textbf{proc}\ P_n(\vec{v_n})\delta_n\ \textbf{end}; \delta\ \}$	Procedures.
$\textbf{if}\ \Phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2$	Synchronized conditional.
$\textbf{while}\ \Phi?\ \textbf{do}\ \delta$	Synchronized loop.
$(\delta_1 \parallel \delta_2)$	Concurrent execution.
$(\delta_1 \rangle\rangle \delta_2)$	Concurrency with different priorities.
$\delta^{\parallel}$	Concurrent iteration.
$< \Phi \rightarrow \delta >$	Interrupt.

a stands for a situation calculus action, Φ stands for a situation calculus formula and δ, possibly subscripted, ranges over ConGolog programs.

3.1 Example

The *emphasized* text is used for constructs of the language; the normal text is used for comments. Lowercase symbols represent constants of the language and uppercase symbols are variables. Predicate and function symbols are lowercase. These conventions will be respected throughout all the paper, unless stated otherwise. We omit comments when it is easy to understand the agent's code.

– Primitive actions:
  ship(Buyer, Product, Required-amount) *send(Sender, Receiver, Message)*
– Situation independent functions:
  min-price(Product) = Min *max-price(Product) = Max*
– Primitive fluents:
  receiving(Sender, Receiver, Message, S)
  Receiver receives *Message* from *Sender* in situation *S*.
  storing(Product, Amount, S)
  The seller stores *Amount* of *Product* in situation *S*.
– Initial state:
  min-price(orange) = 1 *max-price(orange) = 2*
  $\forall$ *S, R, M,* $\neg$ *receiving(S, R, M,* s_0*)* *storing(orange, 1000,* s_0*)*
– Precondition axioms:
  poss(ship(Buyer, Product, Required-amount), S) $\equiv$
    storing(Product, Amount, S) $\wedge$ *Amount* $\geq$ *Required-amount*
  It is possible to ship a certain product if there is enough of this product stored in the department-store.
  poss(send(Sender, Receiver, Message), s) $\equiv$ *true*
  It is always possible to send messages.
– Successor state axioms:

```
poss(A, S) ⟹
    (receiving(Sender, Receiver, Message, do(A, S)) ≡
        (A = send(Sender, Receiver, Message)
        ∨ (A ≠ send(Sender, Receiver, Message) ∧ Message = empty-msg))
```

Receiver is receiving a non-empty *Message* from *Sender* if *Sender* sent *Message* to *Receiver* in the previous situation. Otherwise, *Receiver* picks up an empty message from its mailbox.

```
poss(A, S) ⟹
    (storing(Product, Amount, do(A, S)) ≡
        (A = ship(Buyer, Product, Required-amount) ∧
        storing(Product, Required-amount + Amount, S))
        ∨ (A ≠ ship(Buyer, Product, Required-amount) ∧
        storing(Product, Amount, S)))
```

The seller has a certain *Amount* of *Product* if it had *Required-amount* + *Amount* of *Product* in the previous situation and it shipped *Required-amount* of *Product*, or if it had *Amount* of *Product* in the previous situation and it did not ship any *Product*.

The seller agent executes the procedure *seller-life-cycle* defined in the following way.

```
proc seller-life-cycle
  while true do
   if receiving(Buyer, seller, contractProposal(Product, Required-amount, Price), now)
   then
      if storing(Product, Amount, now) ∧ Amount ≥ Required-amount
          ∧ Price ≥ max-price(Product)
      then ship(Buyer, Product, Required-amount)
          || send(seller, Buyer, accept(Product, Required-amount, Price))
      else .........
   else nil
```

4 AGENT-0

The paper *Agent-Oriented Programming* by Y. Shoham [16] describes the agent-oriented programming (AOP) paradigm, based on the *belief* and *obligation* (or *commitment*) mental categories. A third category, which is not a mental construct, is *capability*. *Decision* (or *choice*) is treated as obligation to oneself. A simple point-based temporal language is used to talk about time. As far as *actions* are concerned, they are not distinguished from facts: the occurrence of an action is represented by the corresponding fact holding. *Beliefs* are represented by means of the modal operator B. The general form of a belief statement is $B_a^t \Phi$ meaning that agent a believes Φ at time t. Modal operators OBL, DEC, CAN and $ABLE$ are used in a similar way to express obligations, decisions, capabilities and abilities (the immediate version of capabilities) of the agent.

In the AGENT-0 programming language [16], the programmer specifies only conditions for making commitments; commitments are actually made and later carried out, automatically at the appropriate times. Commitments are only to primitive actions, those that the agent can directly execute. Their syntax involves basic building blocks as facts (atomic objective sentences), private actions (*(DO t p-action)*), communicative actions (*(INFORM t a fact), (REQUEST t a action), (UNREQUEST t a action)*), nonactions (*(REFRAIN action)*), mental conditions (logical combinations of *mental patterns* of the form *(B fact)* or *((CMT a) action)*), conditional action (*(IF mntlcond action)*) and, finally, message conditions (logical combinations of *message patterns* of the form *(From Type Content)* where *From* is the sender's name, *Type* is *INFORM*, *REQUEST* or *UNREQUEST* and *Content* is a fact statement or an action statement). Finally, the syntax of a *commitment rule* is: *(COMMIT msgcond mntlcond (agent action)*)*, where *msgcond* and *mntlcond* are respectively message and mental conditions, *agent* is an agent name, *action* is an action statement and * denotes repetition of zero or more times. A program is simply a sequence of commitment rules preceded by a definition of the agent's capabilities and initial beliefs, and the fixing of the time grain (this definition will be omitted in our example).

4.1 Example

Variables are preceded by a "?" mark instead of being uppercase, coherently with the language syntax. Universally-quantified variables are denoted by the prefix "?!".

```
CAPABILITIES := ((DO ?time (ship ?!buyer ?prod ?required-amount ?!price))
                    (AND (B (?time (stored ?prod ?stored-amount)))
                       (>= ?stored-amount ?required-amount))
```

The agent has the capability of shipping a certain amount of goods, provided that, at the time of shipping, it believes that such amount is stored in the department-store.

```
INITIAL BELIEFS := (0 (stored orange 1000))      (?!time (min-price orange 1))
                   (?!time (max-price orange 2))
```

```
COMMITMENT RULES := (COMMIT
        (?buyer REQUEST
                (DO now+1 (ship ?buyer ?prod ?req-amnt ?price)))
        (AND (B (now (stored ?prod ?stored-amnt))) (>= ?stored-amnt ?req-amnt)
                     (B (?!time (max-price ?prod ?max))) (>= ?price ?max))
        (?buyer (DO now+1
                (ship ?buyer ?prod ?req-amnt ?price)))
        (myself (INFORM now+1 ?buyer (accepted ?prod ?req-amnt ?price)))
        (myself (DO now+1 (update-product ?prod ?req-amnt))))
```

This commitment rule says that if the seller agent receives a request of shipping a certain amount of goods at a certain price at time *now+1*, and if it believes that at time *now* the required amount is stored in the department-store and the proposed price is greater than *max-price*, the seller agent commits itself to the buyer to ship the goods at time *now+1*, and decides to inform the buyer that its request has been accepted and to update the stored amount of goods.

5 The IMPACT agent language

IMPACT agents [5] are built "on top" of some existing body of code $\mathcal{S}$ specified by the data types or data structures, $\mathcal{T}$, that the agent manipulates and by a set of functions, $\mathcal{F}$, that are callable by external programs. If $\mathtt{f} \in \mathcal{F}$ is an n-ary function defined in the $\mathcal{S}$ package, and $\mathtt{t_1}, \ldots, \mathtt{t_n}$ are *terms* of appropriate types, then $\mathcal{S} : \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is a *code call*. This code call says "Execute function `f` as defined in package $\mathcal{S}$ on the stated list of arguments." A *code call atom* is an expression `cca` of the form $\mathtt{in}(\mathtt{t}, \mathtt{cc})$ or $\mathtt{notin}(\mathtt{t}, \mathtt{cc})$, where `t` is a term and `cc` is a code call. A *code call condition* is a conjunction of code call atoms and *constraint atoms*, which may involve deconstruction operations. The agent has a set of *actions* $\alpha(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables for parameters that can change its state. Every action α has a precondition, a set of effects and a delete list that describe how the agent state changes when the action is executed, and an *execution script or method* consisting of a body of physical code that implements the action. It also has a finite set $\mathcal{IC}$ of *integrity constraints* that the state $\mathcal{O}$ of the agent must satisfy, of the form $\psi \;\Rightarrow\; \chi_a$ where ψ is a code call condition, and χ_a is a code call atom or constraint atom. Finally, each agent has a set of rules called the *agent program* specifying the principles under which the agent is operating. These rules specify, using deontic modalities, what the agent may do ($\mathbf{P}\alpha(\boldsymbol{t})$), must do ($\mathbf{O}\alpha(\boldsymbol{t})$), may not do ($\mathbf{F}\alpha(\boldsymbol{t})$), does ($\mathbf{Do}\,\alpha(\boldsymbol{t})$) and is no more obliged to do ($\mathbf{W}\alpha(\boldsymbol{t})$). $\mathbf{Mod}\alpha(\boldsymbol{t})$ is called *action status atom*, where **Mod** may be **O**, **P**, **F**, **W**, **Do**. If A is an action status atom, then A and $\neg A$ are called *action status literals*. An *agent program* $\mathcal{P}$ is a finite set of rules of the form: $A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n$ where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals.

5.1 Example

We may suppose that the IMPACT program for the seller agent accesses an *oracle* database where information on the stored amount of a certain product and its minimum and maximum price is maintained in a *stored_product* relation; a *msgbox* package that allows agents to exchange messages, as described in Section 3 of [5]; and a mathematical package *math* providing mathematical functions.

- Initial state:
  The *stored_product* relation initially contains the tuple *<orange, 1000, 1, 2>*
- Actions:
  ship(Buyer, Product, Req_amount)

  Pre(ship(Buyer, Product, Req_amount)) =
  in(Old_amount, oracle:select(stored_product.amount, name, =, Product)) $\wedge$
  in(Difference, math:subtract(Old_amount, Req_amount)) $\wedge$
  Difference $\geq$ *0*

  Add(ship(Buyer, Product, Req_amount)) =
  in(Difference, oracle:select(stored_product.amount, name, =, Product))

Del(ship(Buyer, Product, Req_amount)) =
in(Old_amount, oracle:select(stored_product.amount, name, =, Product))
In order for the agent to ship the product, there must be enough of it available (precondition of the action). The effect of shipping is that the amount of available product is updated. We do not provide here the code which realizes this action; we may think that this code issues an order to physically ship the product.

sendMessage(Sender, Receiver, Message)
This action accesses the *msgbox* package putting a tuple in the agent message box. Since the *msgbox* package is discussed in [5], we do not provide further details here.

– Integrity constraints:
in(Min, oracle:select(stored_product.min, name, =, Product)) $\wedge$
in(Max, oracle:select(stored_product.max, name, =, Product)) $\Longrightarrow$
0 $<$ *Min* $<$ *Max*
in(Amount, oracle:select(stored_product.amount, name, =, Product)) $\Longrightarrow$
Amount $>$ *0*

– Agent Program:
Do *sendMessage(Seller, Buyer, accept(Product, Req_amount, Price))* $\leftarrow$
in(⟨ i, Buyer, Seller, contractProposal(Product, Req_amount, Price), T⟩[2]*,*
msgbox:getMessage(Sender)),
in(Max, oracle:select(stored_product.max, name, =, Product)),
in(Amount, oracle:select(stored_product.amount, name, =, Product)),
Price $>=$ *Max, Amount* $>=$ *Req_amount*
This rule says that if all the conditions for accepting a proposal are met then the seller sends a message to the buyer, saying that it accepts the proposal.

O *ship(Buyer, Product, Req_amount)* $\leftarrow$
Do *sendMessage(Seller, Buyer, accept(Product, Req_amount, Price))*
This rule says that if the seller agent accepts the buyer's proposal by sending a message to it, then it is obliged to ship the product.

6 Dylog

The Ph.D. Thesis [14] summarizes the results obtained in the definition of an action language for intelligent agents and describes Dylog, a Prolog implementation of this action language. In the action language each primitive action $a \in A$ is represented by a modality $[a]$. The meaning of the formula $[a]\alpha$ is that α holds after any execution of a. The meaning of the formula $\langle a \rangle \alpha$ is that there is a possible execution of action a after which α holds. There is also a modality $\Box$ which is used to denote those formulae holding in all states. A state consists in a set of *fluents* representing the agent's knowledge

[2] The first element of the tuple says that the message is an input message ("i"); the last element is the reception time.

in that state. The *simple action laws* are rules that allow to describe direct *action laws*, *precondition laws* and *causal laws*.

Action laws define direct effect of primitive actions on a fluent and allow actions with conditional effects to be represented. In Dylog they have the form *a causes F if Fs* where *a* is a primitive action name, *F* is a fluent, and *Fs* is a fluent conjunction, meaning that action *a* has effect on *F*, when executed in a state where the *fluent precondition Fs* holds.

Precondition laws allow *action preconditions*, i.e. those conditions which make an action executable in a state, to be specified. Precondition laws have form *a possible_if Fs* meaning that when a fluent conjunction *Fs* holds in a state, execution of the action *a* is possible in that state.

Causal laws are used to express causal dependencies among fluents and, then, to describe *indirect* effects of primitive actions. They have the form *F if Fs* meaning that the fluent *F* holds if the fluent conjunction *Fs* holds too.

Procedures define the behavior of *complex actions* which are defined on the basis of other complex actions, primitive actions and test actions of the form *?Fs*. In Dylog a procedure is defined as a collection of *procedure clauses* of the form p_0 *is* $p_1, \ldots, p_n$ $(n \geq 0)$ where p_0 is the name of the procedure and p_i, $i = 1 \ldots n$ is either a primitive action, or a test action, a procedure name, or a Prolog goal. Procedures can be recursive and they are executed in a goal directed way, similarly to standard logic programs.

Dylog also provides constructs for planning and sensing that we do not discuss here.

6.1 Example

In the following example, the predicate *is* has its usual meaning in Prolog programs. The Dylog's symbol *is* for defining procedures is substituted with *isp*.

– Functional fluents:
  functionalFluent(storing/2). *functionalFluent(new_message/2).*
  The "/n" after the fluent's name indicates its arity.
– Unchangeable knowledge base (Prolog facts):
  min-price(orange, 1). *max-price(orange, 2).*
– Initial observations:
  obs(storing(orange, 1000)).
– Primitive actions:
  receive.
  This action senses if a fluent *new_message(Sender, Message)* is present in the caller's mailbox. We do not provide further details on its definition, as well as on the *send* action. The interested reader can find it in the Appendix of [14].
  send(Sender, Receiver, Message)
  ship(Buyer, Product, Req_Amnt, Price)
  This action ships the required product to the *Buyer* agent. It is characterized by the following action laws and precondition laws:

```
Action laws:
ship(Buyer, Product, Req_Amnt, Price)
    causes storing(Product, Amount)
    if ?storing(Product, Old_Amount) &
        (Amount is Old_Amount - Req_Amnt).
```

```
Precondition laws:
ship(Buyer, Product, Req_Amnt)
    possible_if ?storing(Product, Old_Amount) &
        (Old_Amount >= Req_Amnt).
ship(Buyer, Product, Req_Amnt)
    possible_if max-price(Product, Max) & (Price >= Max).
```

– Procedures:

```
seller_agent_cycle isp
    receive &
    manage_message &
    seller_agent_cycle.
```

The main cycle for the seller agent consists in waiting for a message, managing it and starting waiting for a message again.

```
manage_message isp
    ?new_message(Buyer, contractProposal(Product, Req_Amnt, Price)) &
    ?storing(Product, Old_Amount) &
    (Old_Amount >= Req_Amnt) &
    max-price(Product, Max) & (Price >= Max) &
    ship(Buyer, Product, Req_Amnt, Price) &
    send(seller, Buyer, accept(Product, Req_Amnt, Price))
```

7 Concurrent METATEM

Concurrent METATEM [6] is a programming language for distributed artificial intelligence based on a linear discrete model of time modeled as an infinite sequence of discrete states which start at the "beginning of time". A Concurrent METATEM system contains a number of concurrently executing agents which are able to communicate through message passing. Each agent executes a first-order temporal logic specification of its desired behavior. Each agent has two main components:

– an *interface* which defines how the agent may interact with its environment (i.e., other agents) by specifying which messages the agent can accept and send;
– a *computational engine*, which defines how the agent may act.

The computational engine of an object is based on the Concurrent METATEM paradigm of executable temporal logics which includes the following operators:

$\psi\,\mathcal{U}\phi$	ψ will be true until ϕ will become true	primitive connective
$\psi\,\mathcal{S}\phi$	ψ was true until ϕ became true	primitive connective
$\bigcirc\phi$	ϕ is true in the next state	[**false** $\mathcal{U}\phi$]
$\circledcirc\phi$	there was a last state, and ϕ was true in the last state	[**false** $\mathcal{S}\phi$]
$\odot\phi$	if there was a last state, then ϕ was true in that state	[$\neg \circledcirc \neg\phi$]
$\Diamond\phi$	ϕ will be true in some future state	[**true** $\mathcal{U}\phi$]
$\blacklozenge\phi$	ϕ was true in some past state	[**true** $\mathcal{S}\phi$]
$\Box\phi$	ϕ will be true in all future states	[$\neg \Diamond\neg\phi$]
$\blacksquare\phi$	ϕ was true in all past states	[$\neg\blacklozenge\neg\phi$]

The idea behind this approach is to directly execute a declarative agent specification given as a set of *program rules* which are temporal logic formulae of the form "antecedent about past $\Rightarrow$ consequent about future".

7.1 Example

- The interface of the *seller* agent is the following:
  seller(contractProposal)[accept, refuse, contractProposal, ship]
  meaning that the seller agent, identified by the *seller* identifier, is able to recognize a *contractProposal* message and is able to broadcast the messages *accept, refuse, contractProposal, ship* (messages include both communicative acts and actions which modify the environment).
- The internal knowledge base of the seller agent contains the following *rigid* predicates (predicates whose value never changes):
  min-price(orange, 1). *max-price(orange, 2).*
- The internal knowledge base of the seller agent contains the following *flexible* predicates (predicates whose value changes over time):
  storing(orange, 1000).
- The program rules of the seller agent are the following ones:
  $\forall$ *Buyer.* $\forall$ *Product.* $\forall$ *Req_Amnt.* $\forall$ *Price.*
  $\circledcirc$*[contractProposal(Buyer, seller, Product, Req_Amnt, Price)* $\wedge$
  storing(Product, Old_Amount) $\wedge$
  Old_Amount $>=$ *Req_Amnt* $\wedge$
  max-price(Product, Max) $\wedge$ *Price* $>=$ *Max]* $\Longrightarrow$
  [ship(Buyer, Product, Req_Amnt, Price) $\wedge$
  accept(seller, Buyer, Product, Req_Amnt, Price)]
  If there exists a previous state where a *Buyer* sent a *contractProposal* message to *seller*, and in that previous state all the conditions were met to accept the *Buyer*'s proposal and ship the required product, then in the current state these actions (shipping and accepting) are performed.

8 Ehhf

The language $\mathcal{E}_{hhf}$ [4] is an executable specification language for modeling concurrent and resource sensitive systems. $\mathcal{E}_{hhf}$ is a multiset-based logic combining features of extensions of logic programming languages like λProlog, e.g. goals with implication and universal quantification, with the notion of *formulae as resources* at the basis of linear logic [9]. A $\mathcal{E}_{hhf}$-program P is a collection of multi-conclusion clauses of the form $A_1 \parr \ldots \parr A_n \multimapinv Goal$, where the A_i are atomic formulae, and the linear disjunction $A_1 \parr \ldots \parr A_n$ corresponds to the head of the clause and $Goal$ is its body. Furthermore, $A \multimapinv B$ is a linear implication. Clauses of this kind *consume* the resources (formulae) they need in order to be applied in a resolution step.

$\mathcal{E}_{hhf}$ provides a way to "guard" the application of a given clause. In the extended type of clauses $G_1 \& \ldots \& G_m \Rightarrow (A_1 \parr \ldots \parr A_n \multimapinv Goal)$, the goal-formulae G_i are *conditions* that must be solved in order for the clause to be triggered.

8.1 Example

- Seller's initial facts:

  min-price(orange, 1). *max-price(orange, 2).*
  storing(orange, 1000). *seller-mail-box([]).*

  We assume that every agent has a mail-box which all the agents in the system can update by calling a *send* predicate. The mailbox of the seller agent is initially empty (*[]*). The reader interested in the definition of communication primitives in $\mathcal{E}_{hhf}$ can find more details in Chapter 5 of [1].

- Seller's life cycle:

  $\forall$ *Message.* $\forall$ *OtherMessages.*
      seller-mail-box([Message|OtherMessages]) $\parr$
      seller-cycle $\multimap$
          manage(Message) $\parr$
          seller-mail-box(OtherMessages) $\parr$
          seller-cycle.

  To satisfy the *seller-cycle* goal, the seller agent must have at least one message in its mail-box. In this case, it consumes the *seller-mail-box([Message| OtherMessages])* and *seller-cycle* goals and produces the new goals of managing the received message (*manage(Message)*), removing it from the mail-box (*seller-mail-box(OtherMessages)*, where the list of messages does not contain *Message* any more) and cycling (*seller-cycle*).

- Seller's rules for managing messages:

  $\forall$ *Buyer.* $\forall$ *Product.* $\forall$ *Req_Amnt.* $\forall$ *Price.*
      Old_Amount >= Req_Amnt &
      difference(Old_Amount, Req_Amnt, Remaining_Amnt) &
      max-price(Product, Max) & *Price >= Max* $\Longrightarrow$
          manage(contractProposal(Buyer, Product, Req_Amnt, Price)) $\parr$
          storing(Product, Old_Amount) $\multimap$
              storing(Product, Remaining_Amount) $\parr$
              ship(Buyer, Product, Req_Amnt, Price) $\parr$
              send(Buyer, accept(seller, Product, Req_Amnt, Price).

  The goals before the $\Longrightarrow$ connective are not consumed by the execution of the rule: they are used to evaluate values (*difference(Old_Amount, Req_Amnt, Remaining_Amnt)*, defined in some way), to compare values (*Old_Amount >= Req_Amnt* and *Price >= Max*) and to get the value of variables appearing in facts that are not changed by the rule (*max-price(Product, Max)*). In this case, they succeed if the conditions for shipping the product are met. The goals *manage(contractProposal(Buyer, Product, Req_Amnt, Price))* and *storing(Product, Old_Amount)* are consumed; they are rewritten in *storing(Product, Remaining_Amount)*, *ship(Buyer, Product, Req_Amnt, Price)* and *send(Buyer, accept(seller, Product, Req_Amnt, Price)*. The *ship* predicate will be defined by some rules that we do not describe here.

9 Desirable features of an "optimal" agent specification language

In this section we compare the agent specification languages introduced so far along different dimensions. The features we think to be mainly relevant for a language for specifying agents and MASs either are related to the basic definition of an agent that we already quoted in the introduction [10] (the first five ones) or are general features that any specification and programming language should have, but that become really indispensable for agent languages. *Time*: agents must either react in a timely fashion to actions taking place in their environment and plan actions in a far future, thus they should be aware of time. *Sensing*: one of the characterizing features of an agent is its ability to sense and perceive the surrounding environment. *Communication*: an agent must be social, namely, it must be able to communicate either with other agents and with human beings. *Concurrency*: agents in a MAS execute autonomously and concurrently and thus it is important that an agent language provides constructs for concurrency among agents (external concurrency) and concurrency within threads internal to the agent (internal concurrency). *Nondeterminism*: the evolution of a MAS consists of a nondeterministic succession of events. *Modularity*: agent programs are typically very complex and a developer would benefit from structuring them by defining modules, macros and procedures. *Semantics*: due to the complexity of languages for agent, providing a clear semantics is the only means to fully understanding the meaning of the constructs they provide and thus exploiting the potentialities of the language.

Since this paper discusses six languages from the perspective of modeling agents and MASs and quickly developing a prototype rather than implementing a working application, we avoid discussing all those technical details which are not fundamental for implementing a prototype, such as efficiency, support to mobility, physical distribution of the agents, integration of external software and traditional programming languages and so on. Even if this paper does not aim at proposing a systematic and general approach to compare agent specification languages, we think that the features we take under consideration can represent a good starting point for evaluating many specification language for MASs.

In the next paragraphs we will analyze the relevant features in detail and in Table 1 we will summarize the results of our comparison. The reader can use this table to discover which language best supports the features she thinks to be mainly relevant for her application. Features which turns to be useful for modeling an application in a certain domain can be unsuitable for modeling another application in a different domain; for this reason we think that any sort of rating of the analyzed features can be provided only on a application-dependent basis and we do not deal with this issue in this paper.

Time. In ConGolog time instants are in direct correspondence with situations: s_0 is the agent's situation at time 0, $do([a_1, ..., a_n], s_0)$ is the agent's situation at time n. However, time is not dealt with explicitly and no operators are provided for managing it. In AGENT-0 time is included in all the constructs of the language. The operations allowed on time variables are only mathematical operations (sums and differences). When programming an agent, it is possible to establish the timegrain of its execution. Time is a central issue in Concurrent METATEM specifications: there are a lot of time-based operators (since, until, in the next state, in the last state, sometimes in the past, sometimes in the future, always in the past, always in the future) which allow to define complex timed expressions. As far as the other languages are concerned, time does not appear in expressions of the language either explicitly or implicitly.

Sensing. Even if Dylog is the only language which provides an explicit construct for defining actions which sense the value of a fluent, all the languages allow to perceive values of atoms that are present in their knowledge base. Whether this knowledge base correctly maintains a model of the environment or not, and thus whether it is possible to "sense" the surrounding environment or not, depends on the given specification. It is worthwhile to note that, in a certain sense, the IMPACT agent programming language is the only one which really senses its (software) environment by means of the code calls mechanism: this mechanism allows an agent to get information by really accessing external software packages, instead of simulating them.

Communication. The languages that embed a support to communication are AGENT-0 and Concurrent METATEM. Among AGENT-0 language constructs, there are the *INFORM*, *REQUEST* and *UNREQUEST* communicative actions which constitute a set of performatives upon which any kind of communication can be built. Unfortunately, for agent A to request an action to agent B it is necessary to know the exact syntax of the action to require. The receiver agent has no means for understanding the content of a request and performing an action consequently, if the action to be performed is not exactly specified as the content of the message itself. This is clearly a strong limitation, which more recent agent communication languages, such as KQML [12] and FIPA ACL [7] have overcome. The same limitation affects Concurrent METATEM: every agent has a communicative interface the other agents in the system must know in order to exchange information. Moreover, Concurrent METATEM does not provide a set of speech acts that all the agents recognize: an AGENT-0 agent may not understand the content of a *REQUEST*, but it at least knows the *REQUEST* performative; this is not true for a Concurrent METATEM agent. The IMPACT language does not provide communication primitives as part of the language, but among the software packages an agent may access there is a *msgbox* package providing message box functionalities. Messages can have any form, adhering to some existing standard or being defined ad-hoc for the application. As far as the other languages are concerned, the specification developer has to define her own communication primitives; however, for all the languages we took under consideration we could easily find examples describing this task.

Concurrency. ConGolog provides different constructs for concurrent execution of processes; these processes may be either internal to a single agent or may represent different agents executing concurrently. Thus, ConGolog supports both concurrency of actions inside an agent and concurrency of agents. The same holds for $\mathcal{E}_{hhf}$, where it is possible to concurrently execute either goals internal to a single agent or goals for activating different agents. As an example of the last case, if different agents were characterized by an *agent-cycle* like the one depicted for the seller agent, it would be possible to prove a goal like *agent1-cycle* || *agent2-cycle* || ... || *agentN-cycle* meaning that *agent1* to *agentN* are executed concurrently. As far as IMPACT is concerned, it associates a body of code implementing a notion of concurrency to each agent in the system, to specify how concurrent actions internal to the agent must be executed. Concurrency of agents is not explicitly specified: the IMPACT language allows the definition of individual agents, not of agent societies. The converse situation takes place with Concurrent METATEM, where concurrency of internal actions is not supported; a Concurrent METATEM specification defines a set of concurrently executing agents which are not able to execute internal concurrent actions. Finally, both Dylog and AGENT-0 do not support any kind of concurrency.

Nondeterminism. ConGolog allows for nondeterministic choice between actions, nondeterministic choice of arguments and nondeterministic iteration. Nondeterminism in the IMPACT language derives from the fact that the feasible, rational and reasonable status sets giving the semantics to agent programs (see the *Semantics* paragraph below) are not unique, thus introducing nondeterminism in the agent's behavior. In Dylog and $\mathcal{E}_{hhf}$ nondeterminism is introduced, as in usual logic programming settings, by the presence of more procedures (rules, in $\mathcal{E}_{hhf}$) defining the same predicate. The main source of nondeterminism in Concurrent METATEM is due to nondeterministic temporal operators such as "sometimes in the past", "sometimes in the future", which do not identify a specific point in time, but may be verified in a range of instants. Finally, AGENT-0 does not seem to support any kind of nondeterministic behavior.

Modularity. All the languages described in this paper support modularity at the agent level, since they allow to define each agent program separately from the definition of the other agents. Both ConGolog and Dylog support the definition of procedures. In ConGolog these procedures are defined by macro expansion into formulae of the situation calculus, while in Dylog they are defined as axioms in the dynamic modal logic. AGENT-0 does not support the definition of procedures, even if in Section 6.3 of [16] macros are used for readability sake. The macro expansion mechanism is not supported by the AGENT-0 implementation. $\mathcal{E}_{hhf}$ supports the definition of procedures as logic programming languages do, by defining rules for solving a goal. Finally, IMPACT and Concurrent METATEM do not allow the definition of procedures.

Semantics. All the languages discussed in this survey, except from AGENT-0, have a formal semantics. Semantics of ConGolog is given as a transition semantics by means of the predicates $Final(\delta, s)$ and $Trans(\delta, s, \delta', s')$. The possible configurations that can be reached by a program δ in situation s are those which are obtained by repeatedly following the transition relation starting from (δ, s) and which are final. There are three different semantics which can be associated to an IMPACT agent program, given its current state and integrity constraints: the feasible, rational and reasonable status set semantics. Reasonable status set semantics is more refined that the rational one, which is more refined that the feasible one. All of them are defined as a set of action status atoms of the form $\mathbf{Do}\,\alpha(\boldsymbol{t})$ that are true with respect to the agent program $\mathcal{P}$, the current state $\mathcal{O}$ and the set $\mathcal{IC}$ of underlying integrity constraints. The logical characterization of Dylog is provided in two steps. First, a multimodal logic interpretation of a dynamic domain description which describes the monotonic part of the language is introduced. Then, an abductive semantics to account for non-monotonic behavior of the language is provided. As far as Concurrent METATEM is concerned, it has a Kripke-style semantics given by the $\models$ relation that assigns the truth value of a formula in a model $\mathcal{M}$ at a particular moment in time i and with respect to a variable assignment. Finally, the $\mathcal{E}_{hhf}$ operational semantics is given by means of a set of rules describing the way sequents can be rewritten. According to the *proof as computation interpretation* of linear logic, sequents represent the state of a computation.

	Time (T)	**Sensing (Sns)**	**Commun. (Cm)**	**Concurr. (Cc)**	**Nondet. (N)**	**Modul. (M)**	**Semantics (Sm)**
ConGolog	⬓	□	□	■	■	■	■
AGENT-0	■	□	■	□	□	□	□
IMPACT	□	■	⬓	■	■	□	■
Dylog	□	⬓	□	□	■	■	■
Conc. METATEM	■	□	■	■	■	□	■
$\mathcal{E}_{hhf}$	□	□	□	■	■	■	■

T: Time is dealt with explicitly (■), implicitly (⬓) or not taken into account (□).
Sns: The language provides constructs for really sensing its software environment (■), for simulating sensing actions (⬓) or no explicit constructs are provided (□).
Cm: The language provides communication primitives embedded in the language (■), it provides a package for communication (⬓), or it does not provide communication primitives but examples of their definition can be easily found (□).
Cc: The language allows to model either internal or external concurrency or both (■) or no kind of concurrency is supported (□).
N: Some kind of nondeterminism is supported (■) or is not supported (□) by the language.
M: The language supports (■) or does not support (□) modularity.
Sm: The language has a formal semantics (■) or as not one (□).

Table 1. Summary of the comparison.

10 Conclusions and future work

Logical languages are suitable to model agents because they allow to easily and intuitively represent mental notions without requiring any special training, allowing to specify agents that have beliefs, plans, goals and that are able to reason about them. The recent organization of workshops fully dedicated to computational logic and multi-agent systems (the CLIMA workshops, `http://mhjcc3-ei.eng.hokudai.ac.jp/clima.html`, `http://research.nii.ac.jp/~ksatoh/clima01.html`, `http://centria.di.fct.unl.pt/~jleite/clima02/`), and some recent papers such as [15] confirm this observation. When logic-based languages are executable like the ones described in this paper, they become a powerful tool for rapid prototyping.

We are firmly convinced that researchers working in the AOSE field can greatly benefit from languages like the ones we described in this paper; they could have even more advantages if all the desired features would be present in the same language or, in alternative, if a subset of these languages could be integrated in a unique common framework, allowing agents defined in different languages to co-exist and interact within the same MAS.

The ARPEGGIO open framework [3], which involves the authors of this paper together with researchers from the University of Maryland, was born with this second aim in mind. The future direction of our work is to implement the ideas behind the ARPEGGIO framework by identifying a suitable subset of languages, among the ones described here, to be integrated within the same environment, and concretely realize this integration.

References

1. A. Aretti. Semantica di sistemi multi-agente in logica lineare. Master's thesis, DISI – Università di Genova, Genova, Italy, 1999. In Italian.
2. P. Ciancarini and M. Wooldridge. Agent-oriented software engineering: The state of the art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*, pages 1–28, Limerick, Ireland, 2000. Springer-Verlag. LNCS 1957.
3. P. Dart, E. Kazmierckaz, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. Combining logical agents with rapid prototyping for engineering distributed applications. In *Proc. 9th International Conference of Software Technology and Engineering (STEP'99)*, Pittsburgh, PA, USA, 1999. IEEE Computer Society Press.
4. G. Delzanno and M. Martelli. Proofs as computations in linear logic. *Theoretical Computer Science*, 258(1–2):269–297, 2001.
5. T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.
6. M. Fisher and H. Barringer. Concurrent METATEM processes – A language for distributed AI. In *Proceedings of the European Simulation Multiconference*, Copenhagen, Denmark, 1991. SCS Press.
7. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Approved for experimental, 14-06-2000, 2000.
8. G. De Giacomo, Y. Lespérance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.
9. J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
10. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
11. V. Mascardi. *Logic-Based Specification Environments for Multi-Agent Systems*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università degli Studi di Genova, Italy, 2002. DISI-TH-2002-04.
12. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *Intelligent Agents II*. Springer Verlag, 1995. LNAI 1037.
13. J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in *Semantic Information Processing*, M. Minsky ed., MIT Press, Cambridge, MA, 1968, pp 110-117.
14. V. Patti. *Programming Rational Agents: a Modal Approach in a Logic Programming Setting*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 2002.
15. F. Sadri and F. Toni. Computational logic and multi-agent systems: a roadmap. Technical report, Department of Computing, Imperial College, London, 1999.
16. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.