TLP **4** (4): 429–494, 2004.
DOI: 10.1017/S1471068404002029 Printed in the United Kingdom

*Logic-based specification languages for intelligent software agents**

VIVIANA MASCARDI, MAURIZIO MARTELLI

DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
(*e-mail:* {mascardi,martelli}@disi.unige.it)

LEON STERLING

Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia
(*e-mail:* leon@cs.mu.oz.au)

Abstract

The research field of Agent-Oriented Software Engineering (AOSE) aims to find abstractions, languages, methodologies and toolkits for modeling, verifying, validating and prototyping complex applications conceptualized as Multiagent Systems (MASs). A very lively research sub-field studies how formal methods can be used for AOSE. This paper presents a detailed survey of six logic-based executable agent specification languages that have been chosen for their potential to be integrated in our ARPEGGIO project, an open framework for specifying and prototyping a MAS. The six languages are ConGolog, AGENT-0, the IMPACT agent programming language, DyLOG, Concurrent METATEM and $\mathcal{E}_{hhf}$. For each executable language, the logic foundations are described and an example of use is shown. A comparison of the six languages and a survey of similar approaches complete the paper, together with considerations of the advantages of using logic-based languages in MAS modeling and prototyping.

KEYWORDS: agent-oriented software engineering, logic-based language, multiagent system

1 Introduction

Today's software applications are typically extremely complex. They may involve heterogeneous components which need to represent their knowledge about the world, about themselves, and about the other entities that populate the world, in order to reason about the world, to plan future actions which should be taken to reach some final goal and to take rapid decisions when the situation demands a quick reaction. Since knowledge and competencies are usually distributed, the components need to interact to exchange information or to delegate tasks. This interaction may follow sophisticated communication protocols. Due to component and system complexity, applications of this kind are difficult to be correctly and efficiently engineered. Indeed a very active research area has been working for almost

* Partially supported by the "Verifica di Sistemi Reattivi Basati su Vincoli (COVER)" project of the Programma di Ricerca Cofinanziato MIUR, Bando 2002, and by the "Discovery" project of the Australian Research Council number DP0209027.

twenty years finding abstractions, languages, methodologies and toolkits for modeling, verifying, validating and finally implementing applications of this kind.

The underlying metaphor is that the components of complex real-world applications are *intelligent agents*. The agents interact, exchanging information and collaborating for reaching a common target, or compete to control some shared resource and to maximize their personal profit, building, in both cases, a society of agents, or *multiagent system* (MAS).

An *intelligent agent*, according to a classical definition proposed by Jennings, Sycara and Wooldridge (Jennings *et al*., 1998) is

> "a computer system, *situated* in some environment, that is capable of *flexible autonomous* actions in order to meet its design objectives.
> Situatedness means that the agent receives sensory input from its environment and that it can perform actions which change the environment in some way.
> By autonomy we mean that the system should be able to act without the direct intervention of humans (or other agents), and that it should have control over its own actions and internal state. [...] By flexible, we mean that the system is:
>
> - responsive: agents should perceive their environment and respond in a timely fashion to changes that occur in it;
> - pro-active: agents should be able to exhibit opportunistic, goal-directed behavior and take the initiative when appropriate;
> - social: agents should be able to interact, when appropriate, with other artificial agents and humans."

Research on Agent-Oriented Software Engineering (AOSE) (Petrie, 2000; Ciancarini and Wooldridge, 2000) aims at providing the means for engineering applications conceptualized as MASs. As pointed out in Ciancarini and Wooldridge (2000), the use of *formal methods* is one of the most active areas in this field, where formal methods play three roles:

- in the *specification* of systems;
- for *directly programming* systems; and
- in the *verification* of systems.

We think that logic-based formal methods can be very effective for fitting all three roles. In fact, the current predominant approach to specifying agents has involved treating the agents as *intentional systems* that may be understood by attributing to them *mental states* such as beliefs, desires and intentions (Dennett, 1987; Wooldridge and Jennings, 1995; Wooldridge, 2000). A number of approaches for formally specifying agents as intention systems have been developed, capable of representing *beliefs, goals* and *actions* of agents and the *ongoing interaction* among them. A large number of logics appear successful at formalizing these concepts in a very intuitive and natural way, including for example *modal logic*, *temporal logic* and *deontic logic*.

Further, there are various logic-based languages for which a working interpreter or an automatic mechanism for animating specifications exists. When these languages are used to specify agents, a working prototype of the given specification is obtained for free and can be used for early testing and debugging of specification. Most of the executable logic-based languages suffer from significant limitations (very low efficiency, poor scalability and modularity, no support for physical distribution of the computation nor for integration of external packages and languages) which make them only suitable for building simple

prototypes. Nevertheless, even if these languages will never be used to build the final application, their execution can give useful and very quick feedback to the MAS developer, who can take advantage of this early testing and debugging for iteratively refining the MAS specification.

Additionally, verification is the process of showing that an implemented system is correct with respect to its original specification. If the language in which the system is implemented is *axiomatizable*, deductive (axiomatic) verification is possible. Otherwise, the *model checking* semantic approach can be followed: given a formula φ of a logic L and a model $\mathscr{M}$ for L, determine whether or not $\mathscr{M} \models_L \varphi$. There are logic-based languages which have been axiomatized, allowing an axiomatic verification, and other languages which can be used for model checking.

Logic-based formalisms are suitable for the stages of specification, direct execution and verification of MAS prototypes. We could ask if some environment and methodology exist that provide a set of logic-based languages for iteratively facing these stages inside a common framework, until a working prototype that behaves as expected is obtained. A preliminary answer can come from ARPEGGIO. ARPEGGIO (*Agent based Rapid Prototyping Environment Good for Global Information Organization* (Dart *et al*., 1999; Zini, 2000; Mascardi, 2002)) is an open framework where specification, execution and verification of MAS prototypes can be carried on choosing the most suitable language or languages from a set of supported ones.

The rationale behind ARPEGGIO is that MAS development requires engineering support for a diverse range of software quality attributes. It is not feasible to create one monolithic AOSE approach to support all quality attributes. Instead, we expect that different approaches will prove suitable to model, verify, or implement different quality attributes. By providing the MAS developer with a large set of languages and allowing the selection of the right language to model, verify or implement each quality attribute, ARPEGGIO goes towards a modular approach to AOSE (Juan *et al*., 2003b; Juan *et al*., 2003a). ARPEGGIO is conceived as the framework providing the building blocks for the development of an hybrid, customizable AOSE methodology. It is not conceived as an hybrid system.

ARPEGGIO draws from three international logic programming research groups: the Logic Programming Group at the Computer Science Department of the University of Maryland, USA; the Logic Programming and Software Engineering Group at the Computer Science & Software Engineering Department of the University of Melbourne, Australia; and the Logic Programming Group at the Computer Science Department of the University of Genova, Italy. An instance of the ARPEGGIO framework, CaseLP (Martelli *et al*., 1999b; Martelli *et al*., 1999a; Marini *et al*., 2000; Zini, 2000; Mascardi, 2002), has been developed and tested on different real-world applications. CaseLP provides a language based on linear-logic, $\mathscr{E}_{hhf}$ (Delzanno, 1997; Delzanno and Martelli, 2001), to specify and verify agent specifications. This language, described in Section 8.1, can also be executed, allowing the direct programming of the code for the prototypical agents. Besides this high-level language, CaseLP provides an extension of Prolog for directly developing the MAS prototype. Although CaseLP demonstrates that the concepts underlying the ARPEGGIO framework can be put into practice and can give interesting results, the set of languages it provides is quite limited. Our motivation behind the development of ARPEGGIO is to

provide a broader set of languages so that the prototype developer can choose the most suitable ones to model and/or program different features of a MAS.

There are two main purposes of this paper. The first is to analyze a set of logic-based languages which have proven useful to specify, execute and/or validate agents and MAS prototypes and which have been integrated or could be integrated into the ARPEGGIO framework. The set we have chosen consists of ConGolog (De Giacomo *et al*., 2000), AGENT-0 (Shoham, 1993), the IMPACT agent language (Eiter *et al*., 1999), DyLOG (Baldoni *et al*., 2000), Concurrent METATEM (Fisher and Barringer, 1991) and $\mathscr{E}_{hhf}$ (Delzanno and Martelli, 2001). These languages have been chosen because, consistent with the ARPEGGIO philosophy, a working interpreter exists for them and they provide useful features for specifying agent systems. Other languages possess these features (see Section 10) and could have been chosen for being analyzed in this paper and for a future integration in ARPEGGIO. However we preferred to provide a focused survey of a small subset of languages rather than a superficial description of a large set. In order to reach a real understanding of the main features of the languages described in this paper we have developed a common running example in all of them.

The second purpose of this paper is to describe the different logics and calculi the executable languages we take into consideration are based on, in order to provide a comprehensive survey on formalisms suitable to model intelligent agents. Some executable languages are based on more than one logic; for example Concurrent METATEM is based on modal and temporal logic, and AGENT-0 is based on modal and deontic logic. The classification we give of agent languages takes into account the predominant logic upon which the language is based.

The structure of the paper is as follows:

- Section 2 describes the running example we will use throughout this paper to practically exemplify the features of the languages we will analyze;
- Section 3 introduces the situation calculus and the ConGolog agent programming language based on it;
- Section 4 discusses modal logic and the AGENT-0 language;
- in Section 5 the main features of deontic logic are shown and the IMPACT programming language is analyzed as an example of an agent language including deontic operators;
- Section 6 discusses dynamic logic and the DyLOG agent language based on it;
- Section 7 describes temporal logic and the Concurrent METATEM agent programming language;
- Section 8 introduces linear logic and analyses the $\mathscr{E}_{hhf}$ language included in the CaseLP instance of the ARPEGGIO framework;
- Section 9 compares the agent programming languages introduced so far based on a set of relevant AOSE features they can support;
- Section 10 discusses related work;
- finally, Section 11 concludes the paper.

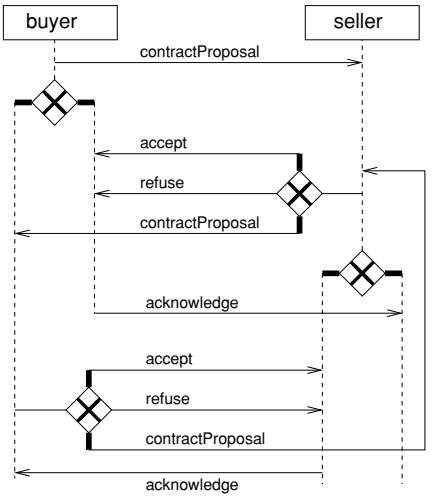

Fig. 1. The contract proposal protocol.

2 Running example

To show how to define an agent in the various agent languages we discuss in this paper, we use a simple example of a seller agent in a distributed marketplace which follows the communication protocol depicted in Figure 1. The notation used in this figure is based on an agent-oriented extension of UML (Odell *et al*., 2000a, Odell *et al*., 2000b); the diamond with the $\times$ inside represents a "xor" connector and the protocol can be repeated more than once (note the bottom arrow from the buyer to the seller labeled with a "contractProposal" message, which loops around and up to point higher up to the seller time line).

The seller agent may receive a contractProposal message from a buyer agent. According to the amount of merchandise required and the price proposed by the buyer, the seller may accept the proposal, refuse it or try to negotiate a new price by sending a contractProposal message back to the buyer. The buyer agent can do the same (accept, refuse or negotiate) when it receives a contractProposal message back from the seller.

The rules guiding the behavior of the seller agent are the following:

> if the received message is contractProposal(merchandise, amount, proposed-price) then
>
> — if there is enough merchandise in the warehouse and the price is greater or equal than a max value, the seller accepts the proposal by sending an accept message to the buyer and concurrently ships the required merchandise to the buyer (if it is not possible to define concurrent actions, answering and shipping merchandise will be executed sequentially);
> — if there is not enough merchandise in the warehouse or the price is lower or equal than a min value, the seller agent refuses the proposal by sending a refuse message to the buyer;
> — if there is enough merchandise in the warehouse and the price is between min and max, the seller sends a contractProposal to the buyer with a proposed

price evaluated as the mean of the price proposed by the buyer and max (we will sometimes omit the definition of this function, which is not of central interest in our example).

In our example, the merchandise to be exchanged are oranges, with minimum and maximum price 1 and 2 euro, respectively. The initial amount of oranges that the seller possesses is 1000.

Our example involves features which fall in the intersection of the six languages and it is therefore quite simple. An alternative choice to providing a simple unifying example would consist of providing six sophisticated examples highlighting the distinguishing features of each of the six languages. However, while sophisticated ad-hoc examples can be found in the papers discussing the six languages, a unifying (though simple) example had not been proposed yet. Consistent with the introductory nature of our paper and with the desire to contribute in an original way to the understanding of the six languages, we opted for the simple unifying example, which is both introductory and original. The about seventy references included in the following six sections should help the reader in finding all the documents she/he needs for deepening her/his knowledge about the six languages discussed in this paper.

3 Situation calculus

The situation calculus (McCarthy, 1963) is well-known in AI research. More recently there have been attempts to axiomatize it. The following description is based upon Pirri and Reiter (1999). $\mathscr{L}_{sit-calc}$ is a second order language with equality. It has three disjoint sorts: *action* for actions, *situation* for situations and a catch-all sort *object* for everything else depending on the domain of application. Apart from the standard alphabet of logical symbols ($\wedge$, $\neg$ and $\exists$, used with their usual meaning), $\mathscr{L}_{sit-calc}$ has the following alphabet:

- Countably infinitely many individual variable symbols of each sort and countably infinitely many predicate variables of all arities.
- Two function symbols of sort *situation*:
  1. A constant symbol S_0, denoting the initial situation.
  2. A binary function symbol $do : action \times situation \rightarrow situation$.
     $do(a, s)$ denotes the successor situation resulting from performing action a in situation s.
- A binary predicate symbol $\sqsubset$: $situation \times situation$, defining an ordering relation on situations. The intended interpretation of situations is as action histories, in which case $s \sqsubset s'$ means that s' can be reached by s by a finite application of actions.
- A binary predicate symbol $Poss : action \times situation$. The intended interpretation of $Poss(a, s)$ is that it is possible to perform the action a in the situation s.
- Countably infinitely many predicate symbols used to denote situation independent relations and countably infinitely many function symbols used to denote situation independent functions.
- A finite or countably infinite number of function symbols called *action functions* and used to denote actions.

- A finite or countably infinite number of *relational fluents* (predicate symbols used to denote situation dependent relations).
- A finite or countably infinite number of function symbols called *functional fluents* and used to denote situation dependent functions.

In the axiomatization proposed in Levesque *et al*. (1998), axioms are divided into domain axioms and domain independent foundational axioms for situations. Besides axioms, Levesque *et al*. (1998) also introduce basic theories of actions and a metatheory for the situation calculus which allows to determine when a basic action theory is satisfiable and when it entails a particular kind of sentence, called *regressable sentences*. Here we only discuss domain independent foundational axioms for situations. Since the scope of this paper is to provide introductory material which can be understood with little effort, we will address neither domain axioms nor the metatheory for the situation calculus, both of which require a strong technical background.

3.1 Foundational axioms for situations

There are four foundational axioms for the situation calculus, based on Pirri and Reiter (1999), but simpler that the ones presented there. They capture the intuition that situations are finite sequences of actions where the second order induction principle holds, and that there is a "subsequence" relation among them. In the following axioms, P is a predicate symbol.

$$do(a_1, s_1) = do(a_2, s_2) \Rightarrow a_1 = a_2 \wedge s_1 = s_2 \tag{1}$$

$$\forall P \cdot P(S_0) \wedge \forall a, s \cdot [P(s) \Rightarrow P(do(a, s))] \Rightarrow \forall s \cdot P(s) \tag{2}$$

Axiom 1 is a unique name axiom for situations: two situations are the same iff they are the same sequence of actions. Axiom 2 is second order induction on situations. The third and fourth axioms are:

$$\neg (s \sqsubset S_0) \tag{3}$$

$$(s \sqsubset do(a, s')) \equiv (s \sqsubseteq s') \tag{4}$$

Here $s \sqsubseteq s'$ is an abbreviation for $(s \sqsubset s') \vee (s = s')$. The relation $\sqsubset$ provides an ordering relation on situations. Intuitively, $s \sqsubset s'$ means that the situation s' can be obtained from the situation s by adding one or more actions to the end of s.

The above four axioms are *domain independent*. They provide the basic properties of situations in any domain specific axiomatization of particular fluents and actions.

3.2 ConGolog

ConGolog is a concurrent programming language based on the situation calculus which includes facilities for prioritizing the concurrent execution, interrupting the execution when certain conditions become true, and dealing with exogenous actions. As stated by De Giacomo, Lespérance and Levesque (De Giacomo *et al*., 2000), the adoption of a language like ConGolog is a promising alternative to traditional plan synthesis, since it allows high-level program execution. ConGolog is an extension of the programming language Golog

(Levesque *et al.*, 1997): in Section 3.2.1 we present Golog and in Section 3.2.2 we deal with its extension ConGolog.

3.2.1 Golog

Golog is a logic-programming language whose primitive actions are drawn from a background domain theory.

Golog programs are inductively defined as:

- Given a situation calculus action a with all situation arguments in its parameters replaced by the special constant *now*, a is a Golog program (primitive action).
- Given a situation calculus formula φ with all situation arguments in its parameters replaced by the special constant *now*, $\varphi?$ is a Golog program (wait for a condition).
- Given δ, δ_1, δ_2, δ_n Golog programs,
  - $(\delta_1 ; \delta_2)$ is a Golog program (sequence);
  - $(\delta_1 \mid \delta_2)$ is a Golog program (nondeterministic choice between actions);
  - $\pi v \cdot \delta$ is a Golog program (nondeterministic choice of arguments);
  - δ^* is a Golog program (nondeterministic iteration);
  - $\{\textbf{proc}\ P_1(\vec{v_1})\delta_1\ \textbf{end}; \ldots \textbf{proc}\ P_n(\vec{v_n})\delta_n\ \textbf{end};\ \delta\ \}$ is a Golog program (procedure: P_i are procedure names and v_i are their parameters).

Program Execution. Given a domain theory $\mathscr{D}$ and a program δ the execution task is to find a sequence $\vec{a}$ of actions such that:

$$\mathscr{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$$

where

$$Do(\delta, s, s')$$

means that program δ when executed starting in situation s has s' as a legal terminating situation, and

$$do(\vec{a}, s) = do([a_1, \ldots, a_n], s)$$

is an abbreviation for

$$do(a_n, do(a_{n-1}, \ldots, do(a_1, s))).$$

Since Golog programs can be nondeterministic, there may be several terminating situations for the same program and starting situation. $Do(\delta, s, s')$ is formally defined by means of the following inductive definition:

1. Primitive actions ($a[s]$ denotes the action obtained by substituting the situation variable s for all occurrences of *now* in functional fluents appearing in a):

$$Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$$

2. Wait/test actions ($\varphi[s]$ denotes the formula obtained by substituting the situation variable s for all occurrences of *now* in functional and predicate fluents appearing in φ):

$$Do(\varphi?, s, s') \stackrel{def}{=} \varphi[s] \wedge s = s'$$

3. Sequence:

$$Do(\delta_1;\delta_2,s,s') \stackrel{def}{=} \exists s'' \cdot Do(\delta_1,s,s'') \wedge Do(\delta_2,s'',s')$$

4. Nondeterministic branch:

$$Do(\delta_1 \mid \delta_2,s,s') \stackrel{def}{=} Do(\delta_1,s,s') \vee Do(\delta_2,s,s')$$

5. Nondeterministic choice of argument ($\pi x \cdot \delta(x)$ is executed by nondeterministically picking an individual x, and for that x, performing the program $\delta(x)$):

$$Do(\pi x \cdot \delta(x),s,s') \stackrel{def}{=} \exists x \cdot Do(\delta(x),s,s')$$

6. Nondeterministic iteration:

$$Do(\delta^*,s,s') \stackrel{def}{=} \forall P \cdot \{\forall s_1 \cdot P(s_1,s_1) \wedge \forall s_1,s_2,s_3 \cdot [P(s_1,s_2) \wedge Do(\delta,s_2,s_3) \Rightarrow P(s_1,s_3)]\} \Rightarrow P(s,s')$$

   P is a binary predicate symbol. Saying "(x,x') is in the set (defined by P)" is equivalent to saying "$P(x,x')$ is true". Doing action δ zero or more times leads from the situation s to the situation s' if and only if (s,s') is in every set (and therefore, the smallest set) such that:

   (a) (s_1,s_1) is in the set for all situations s_1.
   (b) Whenever (s_1,s_2) is in the set, and doing δ in situation s_2 leads to situation s_3, then (s_1,s_3) is in the set.

   The above is the standard second order definition of the set obtained by nondeterministic iteration.

We do not deal with expansion of procedures. The reader can see Levesque *et al*. (1997) for the details.

3.2.2 ConGolog

ConGolog is an extended version of Golog that incorporates concurrency, handling:

- concurrent processes with possibly different priorities;
- high-level interrupts and
- arbitrary exogenous actions.

ConGolog programs are defined by the following inductive rules:

- All Golog programs are ConGolog programs.
- Given a situation calculus formula φ with all situation arguments in its parameters replaced by the special constant *now*, and δ, δ_1, δ_2 ConGolog programs,
  - — **if** φ **then** δ_1 **else** δ_2 is a ConGolog program (synchronized conditional);
  - — **while** $\varphi?$ **do** δ is a ConGolog program (synchronized loop);
  - — $(\delta_1 \parallel \delta_2)$ is a ConGolog program (concurrent execution);
  - — $(\delta_1 \rangle\rangle \delta_2)$ is a ConGolog program (concurrency with different priorities);
  - — $\delta^{\parallel}$ is a ConGolog program (concurrent iteration);
  - — $\langle \varphi \rightarrow \delta \rangle$ is a ConGolog program (interrupt).

The constructs **if** φ **then** δ_1 **else** δ_2 and **while** $\varphi?$ **do** δ are the synchronized versions of the usual if-then-else and while-loop. They are synchronized in the sense that the test of the condition φ does not involve a transition per se: the evaluation of the condition and the first action of the branch chosen will be executed as an atomic action. The construct $(\delta_1 \parallel \delta_2)$ denotes the concurrent execution of the actions δ_1 and δ_2. $(\delta_1 \rangle\rangle \delta_2)$ denotes the concurrent execution of the actions δ_1 and δ_2 with δ_1 having higher priority than δ_2, restricting the possible interleavings of the two processes: δ_2 executes only when δ_1 is either done or blocked. The construct $\delta^{\parallel}$ is like nondeterministic iteration, but where the instances of δ are executed concurrently rather than in sequence. Finally, $\langle \varphi \rightarrow \delta \rangle$ is an interrupt. It has two parts: a trigger condition φ and a body δ. The idea is that the body δ will execute some number of times. If φ never becomes true, δ will not execute at all. If the interrupt gets control from higher priority processes when φ is true, then δ will execute. Once it has completed its execution, the interrupt is ready to be triggered again. This means that a high priority interrupt can take complete control of the execution.

3.2.3 Semantics

The semantics of Golog and ConGolog is in the style of transition semantics. Two predicates are defined which say when a program δ can legally terminate in a certain situation s ($Final(\delta, s)$) and when a program δ in the situation s can legally execute one step, ending in situation s' with program δ' remaining ($Trans(\delta, s, \delta', s')$). *Final* and *Trans* are characterized by a set of equivalence axioms, each depending on the structure of the first argument. To give the flavor of how these axioms look like, we show the ones for empty program *nil*, atomic action a, testing $\varphi?$, nondeterministic branch $(\delta_1 \mid \delta_2)$ and concurrent execution $(\delta_1 \parallel \delta_2)$[1]. The reader can find the complete set of axioms for *Final* and *Trans* in De Giacomo *et al*. (2000).

$$
\begin{array}{lcl}
Trans(nil, s, \delta', s') & \equiv & \text{false} \\
Trans(a, s, \delta', s') & \equiv & Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\
Trans(\varphi?, s, \delta', s') & \equiv & \varphi[s] \wedge \delta' = nil \wedge s' = s \\
Trans(\delta_1 \mid \delta_2, s, \delta', s') & \equiv & Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\delta_1 \parallel \delta_2, s, \delta', s') & \equiv & \\
\quad \exists \gamma \cdot \delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee & & \\
\quad \exists \gamma \cdot \delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s') & &
\end{array}
$$

The meaning of these axioms is that: (nil, s) does not evolve to any configuration; (a, s) evolves to $(nil, do(a[s], s))$ provided that $a[s]$ is possible in s; $(\varphi?, s)$ evolves to (nil, s) provided that $\varphi[s]$ holds; $(\delta_1 \mid \delta_2, s)$ can evolve to (δ', s') provided that either (δ_1, s) or (δ_2, s) can do so; and finally, $(\delta_1 \parallel \delta_2, s)$ can evolve if (δ_1, s) can evolve and δ_2 remains unchanged or (δ_2, s) can evolve and δ_1 remains unchanged.

[1] In order to define axioms properly, programs should be encoded as first order terms. We avoid dealing with this encoding, and describe axioms as if programs were already first order terms.

$$
\begin{array}{lcl}
Final(nil, s) & \equiv & \text{true} \\
Final(a, s) & \equiv & \text{false} \\
Final(\varphi?, s) & \equiv & \text{false} \\
Final(\delta_1 \mid \delta_2, s) & \equiv & Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\delta_1 \parallel \delta_2, s) & \equiv & Final(\delta_1, s) \wedge Final(\delta_2, s)
\end{array}
$$

These axioms say that (nil, s) is a final configuration while neither (a, s) nor $(\varphi?, s)$ are. $(\delta_1 \mid \delta_2, s)$ is final if either (δ_1, s) is or (δ_2, s) is, while $(\delta_1 \parallel \delta_2, s)$ is final if both (δ_1, s) and (δ_2, s) are.

The possible configurations that can be reached by a program δ in situation s are those obtained by repeatedly following the transition relation denoted by *Trans* starting from (δ, s). The reflexive transitive closure of *Trans* is denoted by *Trans**. By means of *Final* and *Trans** it is possible to give a new definition of *Do* as

$$Do(\delta, s, s') \stackrel{def}{=} \exists \delta' \cdot Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

3.2.4 Implementation

A simple implementation of ConGolog has been developed in Prolog. The definition of the interpreter is lifted directly from the definitions of *Final*, *Trans* and *Do* given above. The interpreter requires that the program's precondition axioms, successor state axioms and axioms about the initial situation be expressible as Prolog clauses. In particular, the usual *closed world assumption* is made on the initial situation. Section 8 of De Giacomo *et al*. (2000) describes the ConGolog interpreter in detail and proves its correctness under suitable assumptions. The interpreter is included into a more sophisticated toolkit which provides facilities for debugging ConGolog programs and delivering process modeling applications by means of a graphical interface. Visit the Cognitive Robotics Group Home Page (2002) to download the interpreter of both ConGolog and its extensions discussed in the next section.

3.2.5 Extensions

Some variants of ConGolog have been developed in the last years:

- Legolog (*LEGO MINDSTORM in (Con)Golog* (Levesque and Pagnucco, 2000)) uses a controller from the Golog family of planners to control a MINDSTORM robot. Legolog is capable of dealing with primitive actions, exogenous actions and sensing; the Golog controller is replaced with an alternate planner. Visit the Legolog Home Page (2000) for details.
- IndiGolog (*Incremental Deterministic (Con)Golog* (De Giacomo *et al*., 2002)) is a high-level programming language where programs are executed incrementally to allow for interleaved action, planning, sensing, and exogenous events. IndiGolog provides a practical framework for real robots that must react to the environment and continuously gather new information from it. To account for planning, IndiGolog provides a local lookahead mechanism with a new language construct called *the search operator*.

- CASL (*Cognitive Agent Specification Language* (Shapiro *et al.*, 2002)) is a framework for specifying complex MASs which also provides a verification environment based on the PVS verification system (Owre *et al.*, 1996).
- A class of knowledge-based Golog programs is extended with sense actions in Reiter (2001).

Most of the publications on Golog, ConGolog and their extensions can be found at the Cognitive Robotics Group Home Page (2002).

3.2.6 Example

The program for the seller agent, written in ConGolog, could look as follows. 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 (thus, the *Poss* predicate symbol introduced in the beginning of Section 3 is written *poss* in the example). These Prolog-like conventions will be respected in all the examples appearing in the paper, unless stated otherwise.

- Primitive actions declaration:

  ship(Buyer, Merchandise, Required-amount)
  The seller agent delivers the *Required-amount* of *Merchandise* to the *Buyer*.
  send(Sender, Receiver, Message)
  Sender sends *Message* to *Receiver*.

- Situation independent functions declaration:

  min-price(Merchandise) = Min
  The minimum price the seller is willing to take under consideration for *Merchandise* is *Min*.
  max-price(Merchandise) = Max
  The price for *Merchandise* that the seller accepts without negotiation is equal or greater than *Max*.

- Primitive fluents declaration:

  receiving(Sender, Receiver, Message, S)
  Receiver receives *Message* from *Sender* in situation *S*.
  storing(Merchandise, Amount, S)
  The seller stores *Amount* of *Merchandise* in situation *S*.

- Initial situation axioms:

  $min\text{-}price(orange) = 1$
  $max\text{-}price(orange) = 2$
  $\forall\, S, R, M.\ \neg\, receiving(S, R, M, s_0)$
  $storing(orange, 1000, s_0)$

- Precondition axioms:

  $poss(ship(Buyer, Merchandise, Required\text{-}amount), S) \equiv$
  $\quad storing(Merchandise, Amount, S) \wedge Amount \geqslant Required\text{-}amount$

It is possible to ship merchandise iff there is enough merchandise stored in the warehouse.

$poss(send(Sender, Receiver, Message), S) \equiv true$

It is always possible to send messages.

- Successor state axioms:

  $receiving(Sender, Receiver, Message,$
  $\quad do(send(Sender, Receiver, Message), S)) \equiv true$

  Receiver receives *Message* from *Sender* in $do(send(Sender, Receiver, Message), S)$ reached by executing $send(Sender, Receiver, Message)$ in *S*. For sake of conciseness we opted for a very simple formalization of agent communication. More sophisticated formalizations can be found in (Marcu *et al*., 1995) and (Shapiro *et al*., 1998).

  $storing(Merchandise, Amount, do(A, S)) \equiv$
  $\quad (A = ship(Buyer, Merchandise, Required\text{-}amount) \wedge$
  $\quad storing(Merchandise, Required\text{-}amount + Amount, S))$
  $\quad \vee (A \neq ship(Buyer, Merchandise, Required\text{-}amount) \wedge$
  $\quad storing(Merchandise, Amount, S))$

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

We may think that a buyer agent executes a *buyer-life-cycle* procedure concurrently with the seller agent procedure *seller-life-cycle*. *buyer-life-cycle* defines the actions the buyer agent takes according to its internal state and the messages it receives. The *seller-life-cycle* is defined in the following way.

proc *seller-life-cycle*

while *true* **do**
if $receiving(Buyer, seller, contractProposal(Merchandise, Required\text{-}amount, Price), now)$
then
if $storing(Merchandise, Amount, now)$
$\wedge\ Amount \geqslant Required\text{-}amount$
$\wedge\ Price \geqslant max\text{-}price(Merchandise)$
then $ship(Buyer, Merchandise, Required\text{-}amount)$
$\parallel\ send(seller, Buyer, accept(Merchandise, Required\text{-}amount, Price))$
else
if $(storing(Merchandise, Amount, now)$
$\wedge\ Amount < Required\text{-}amount)$
$\vee\ Price \leqslant min\text{-}price(Merchandise)$
then $send(seller, Buyer, refuse(Merchandise, Required\text{-}amount, Price))$
else

if *storing(Merchandise, Amount, now)*
$\wedge$ *Amount* $\geqslant$ *Required-amount*
$\wedge$ *min-price(Merchandise)* $<$ *Price* $<$ *max-price(Merchandise)*
then *send(seller, Buyer,*
contractProposal(Merchandise, Required-amount,
(Price+max-price(Merchandise))/2))
else nil
else nil

4 Modal logic

This introduction is based on Fisher and Owens (1995). Modal logic is an extension of classical logic with (generally) a new connective $\Box$ and its derivable counterpart $\Diamond$, known as *necessity* and *possibility*, respectively. If a formula $\Box p$ is true, it means that p is necessarily true, i.e. true in every possible scenario, and $\Diamond p$ means that p is possibly true, i.e. true in at least one possible scenario. It is possible to define $\Diamond$ in terms of $\Box$:

$$\Diamond p \Leftrightarrow \neg\Box\neg p$$

so that p is possible exactly when its negation is not necessarily true. In order to give meaning to $\Box$ and $\Diamond$, models for modal logic are usually based on *possible worlds*, which are essentially a collection of connected models for classical logic. The possible worlds are linked by a relation which determines which worlds are accessible from any given world. It is this *accessibility relation* which determines the nature of the modal logic. Each world is given a unique label, taken from a set S, which is usually countably infinite. The accessibility relation R is a binary relation on S. The pairing of S and R defines a *frame* or structure which underpins the model of modal logic. To complete the model we add an interpretation

$$h : S \times PROP \rightarrow \{\text{true}, \text{false}\}$$

of propositional formulae $\in PROP$ in each state.
Given $s \in S$ and $a \in PROP$,

$$\langle S, R, h\rangle \models_s a \qquad \text{iff} \quad h(s, a) = true$$

This is read as: a is true in world s in the model $\langle S, R, h\rangle$ iff h maps a to true in world s. In general when a formula φ is true in a world s in a model $\mathscr{M}$, it is denoted by

$$\mathscr{M} \models_s \varphi$$

and if it is true in every world in the set S, it is said to be true in the model, and denoted by

$$\mathscr{M} \models \varphi$$

The boolean connectives are given the usual meaning:

$$\langle S, R, h\rangle \models_s \varphi \vee \psi \qquad \text{iff} \quad \langle S, R, h\rangle \models_s \varphi \ \text{ or } \ \langle S, R, h\rangle \models_s \psi$$
$$\langle S, R, h\rangle \models_s \varphi \Rightarrow \psi \qquad \text{iff} \quad \langle S, R, h\rangle \models_s \varphi \ \text{ implies } \ \langle S, R, h\rangle \models_s \psi$$

The frame enters the semantic definition only when the modality $\Box$ is used, as the formula $\Box\varphi$ is true in a world s exactly when every world t in S which is accessible from s (i.e. such that $s\ R\ t$) has φ true. More formally,

$$\langle S, R, h\rangle \models_s \Box\varphi \quad \text{iff} \quad \text{for all} \ \ t \in S, \quad s\ R\ t \ \text{ implies } \ \langle S, R, h\rangle \models_t \varphi$$

The models $\langle S, R, h\rangle$ and the semantics we introduced for connectives are also known as Kripke models (or structures) and Kripke semantics, respectively (Kripke, 1963b; Kripke, 1963a; Kripke, 1965), from the name of the author who mainly contributed to developing a satisfactory semantic theory of modal logic.

4.1 AGENT-0

Shoham's paper *Agent-Oriented Programming* (Shoham, 1993) is one of the most cited papers in the agent community, since it proposed a new programming paradigm that

> promotes a societal view of computation, in which multiple "agents" interact with one another.

In this section we first introduce the basic concepts of the agent-oriented programming (AOP) paradigm, and then we present the AGENT-0 programming language. This is often referred to as the first agent programming language, even though

> the simplifications embodied in AGENT-0 are so extreme that it may be tempting to dismiss it as uninteresting (Shoham, 1993).

We opted for describing AGENT-0 as the language representing the class of languages based on mental modalities because it was the first one to adopt this approach. Other agent programming languages including mental modalities are 3APL (Hindriks *et al.*, 1998a) and AgentSpeak(L) (Rao, 1996) which will be discussed in Section 10.

For Shoham, a complete AOP system will include three primary components:

1. A restricted formal language with clear syntax and semantics for describing mental states; the mental state will be defined uniquely by several modalities, such as belief and commitments.
2. An interpreted programming language in which to define and program agents, with primitive commands such as *REQUEST* and *INFORM*.
3. An "agentification process" to treat existing hardware devices or software applications like agents.

The focus of Shoham's work is on the second component.

The mental categories upon which the AOP is based are *belief* and *obligation* (or *commitment*). A third category, which is not a mental construct, is *capability*. *Decision* (or *choice*) is treated as obligation to oneself.

Since *time* is basic to the mental categories, it is necessary to specify it. A simple point-based temporal language is used to talk about time; a typical sentence will be

$$holding(robot, cup)^t$$

meaning that the robot is holding the cup at time t.

As far as *actions* are concerned, they are not distinguished from facts: the occurrence of an action is represented by the corresponding fact being true.

Beliefs are represented by means of the modal operator B. The general form of a belief statement is

$$B_a^t \varphi$$

meaning that agent a believes φ at time t. φ may be a sentence like $holding(robot, cup)^t$ or a belief statement: nested belief statements like $B_a^3 B_b^{10} like(a, b)^7$, meaning that at time 3 agent a believes that at time 10 agent b will believe that at time 7 a liked b, are perfectly legal in the AOP language.

The fact that at time t an agent a commits himself to agent b about φ is represented by the sentence

$$OBL_{a,b}^t \varphi$$

A *decision* is an obligation to oneself, thus

$$DEC_a^t f \stackrel{def}{=} OBL_{a,a}^t \varphi$$

The fact that at time t agent a is *capable* of φ is represented by

$$CAN_a^t \varphi$$

Finally, there is an "immediate" version of *CAN*:

$$ABLE_a \varphi \stackrel{def}{=} CAN_a^{time(\varphi)} \varphi$$

where $time(B_a^t \psi) = t$ and $time(pred(arg_1, \ldots, arg_n)^t) = t$.

To allow the modalities introduced so far resemble their common sense counterparts, some assumptions are made:

- *Internal consistency*: both the beliefs and the obligations are assumed to be internally consistent.
- *Good faith*: agents commit only to what they believe themselves capable of, and only if they really mean it.
- *Introspection*: agents are aware of their obligations.
- *Persistence of mental state*: agents have perfect memory of, and faith in, their beliefs, and only let go off a belief if they learn a contradictory fact. Obligations too should persist, and capabilities too tend not to fluctuate wildly.

AGENT-0 is a simple programming language that implements some of the AOP concepts described above. Since AGENT-0 allows to define one program for each agent involved in the system, it is no longer necessary to explicitly say which agent is performing which action; as an example, the statement $B_a^t \varphi$ becomes $(B(t(\varphi)))$ in the body of code associated with agent a. In AGENT-0 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. Before defining the syntax of commitments, other basic definitions are necessary.

Facts. Fact statements constitute a tiny fragment of the temporal language described in the previous paragraph: they are essentially the atomic objective sentences of the form

(t atom)

and

(NOT (t atom))

For example, *(0 (stored orange 1000))* is an AGENT-0 fact stating that at time 0 there where 1000 oranges in the warehouse.

Private actions. The syntax for private actions is

(DO t p-action)

where *t* is a time point and *p-action* is a private action name. The effects of private actions may or may not be visible to other agents.

Communicative actions. There are three types of communicative actions:

(INFORM t a fact)

where *t* is the time point in which informing takes place, *a* is the receiver's name and *fact* is a fact statement.

(REQUEST t a action)

where *t* is a time point, *a* is the receiver's name and *action* is an action statement.

(UNREQUEST t a action)

where *t* is a time point, *a* is the receiver's name and *action* is an action statement.

Nonaction. A "nonaction" prevents an agent from committing to a particular action.

(REFRAIN action)

Mental conditions. A mental condition is a logical combination of *mental patterns* which may assume two forms:

(B fact)

meaning that the agent believes *B* or

((CMT a) action)

where *CMT* stands for commitment. The information about time is included in facts and actions; an example of a mental pattern is *(B (3 (stored orange 950)))* meaning that the agent believes that at time 3 there were 950 oranges left in the warehouse.

Capabilities. The syntax of a capability is

(action mentalcondition)

meaning that the agent is able to perform *action* provided that *mentalcondition* (see above) is true. Throughout the example we will use this syntax which is inherited from the original paper, even if a notation including the *CAN* keyword (namely, *(CAN action mentalcondition)*) would be more appropriate.

Conditional action. The syntax of a conditional action is

(IF mentalcondition action)

meaning that *action* can be performed only if *mentalcondition* holds.

Message condition. A message condition is a logical combination of *message patterns*, which are triples

(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.

Commitment rule. A commitment rule has the form:

(COMMIT messagecondition mentalcondition (agent action))*

where *messagecondition* and *mentalcondition* are respectively message and mental conditions, *agent* is an agent name, *action* is an action statement and * denotes repetition of zero or more times. The intuition behind the commitment rule *(COMMIT msgcond mntcond* $(ag_1\ act_1) \ldots (ag_n\ act_n)$*)* in the program defining the behavior of agent *ag* is that if *ag* receives a message satisfying *msgcond* and its mental state verifies the condition *mntcond*, it commits to agent ag_1 about $act_1, \ldots$, and to agent ag_n about act_n. Note that commitment rules are identified by the *COMMIT* keyword and commitment mental pattern (see the definition of mental conditions above) are identified by the *CMT* keyword. We adopt this syntax to be consistent with the original paper even if we are aware that using similar keywords for different syntactic objects may be confusing.

Program. A program is defined by the time unit, called "timegrain", followed by the capabilities, the initial beliefs and the commitment rules of an agent. Timegrain ranges over *m* (minute), *h* (hour), *d* (day) and *y* (year).

4.1.1 Semantics

No formal semantics for the language is given.

4.1.2 Implementation

A prototype AGENT-0 interpreter has been implemented in Common Lisp and has been installed on Sun/Unix, DecStation/Ultrix and Macintosh computers. Both the interpreter and the programming manual are available to the scientific community. A separate implementation has been developed by Hewlett Packard as part of a joint project to incorporate AOP in the New Wave^{TM} architecture.

The AGENT-0 engine is characterized by the following two-step cycle:

1. Read the current messages and update beliefs and commitments.
2. Execute the commitments for the current time, possibly resulting in further belief change.

Actions to which agents can be committed include communicative ones such as informing and requesting, as well as arbitrary private actions.

4.1.3 Extensions

Two extensions of AGENT-0 have been proposed:

- PLACA (Thomas, 1995) enriches AGENT-0 with a mechanism for flexible management of plans. It adds two data structures to the agent's state: a list of intentions and a list of plans. Intentions are adopted is a similar way that commitments are; the PLACA command *(ADOPT (INTEND x))* means that the agent will add the intention to do *x* to its intention list. Plans are created by an external plan generator to meet

these intentions. This approach gives the system the ability to dynamically alter plans that are not succeeding.

- Agent-K (Davies and Edwards, 1994) is an attempt to standardize the message passing functionality in AGENT-0. It combines the syntax of AGENT-0 (without support for the planning mechanisms of PLACA) with the format of KQML (*Knowledge Query and Manipulation Language* (Mayfield *et al.*, 1995)) to ensure that messages written in languages different from AGENT-0 can be handled. Agent-K introduces two major changes to the structure of AGENT-0: first, it replaces outgoing *INFORM*, *REQUEST*, and *UNREQUEST* message actions with one command, *KQML*, that takes as its parameters the message, the time, and the *KQML* type; second, it allows many commitments to match a single message. In AGENT-0 the multiple commitment mechanism was not defined and the interpreter simply selected the first rule that matched a message.

4.1.4 Example

The AGENT-0 program for the seller agent may be as follows. Variables are preceded by a "?" mark instead of being uppercase, coherently with the language syntax. Universally quantified variables, whose scope is the entire formula, are denoted by the prefix "?!".

timegrain := m

The program is characterized by a time-grain of one minute.

CAPABILITIES := ((DO ?time (ship ?!buyer ?merchandise ?required-amount ?!price))
(AND (B (?time (stored ?merchandise ?stored-amount)))
(⩾ ?stored-amount ?required-amount)))

The agent has the capability of shipping a certain amount of merchandise, provided that, at the time of shipping, it believes that such amount is stored in the warehouse.

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

The agent has initial beliefs about the minimum price, maximum price and stored amount of oranges. The initial belief about stored oranges only holds at time 0, since this amount will change during the agent's life, while beliefs about minimum and maximum prices hold whatever the time.

COMMITMENT RULES := (COMMIT
(?buyer REQUEST
(DO now (ship ?buyer ?merchandise ?req-amnt ?price)))

(AND (B (now (stored ?merchandise ?stored-amount)))
(⩾ ?stored-amount ?req-amnt)

```
                    (B (now (max-price ?merchandise ?max)))
                    (⩾ ?price ?max))

(?buyer (DO now
            (ship ?buyer ?merchandise ?req-amnt ?price)))
(myself (INFORM now ?buyer
            (accepted ?merchandise ?req-amnt ?price)))
(myself (DO now (update-merchandise ?merchandise ?req-amnt)))
)
```

The first commitment rule says that if the seller agent receives a request of shipping a certain amount of merchandise at a certain price, and if it believes that the required amount is stored in the warehouse and the proposed price is greater than *max-price*, the seller agent commits itself to the buyer to ship the merchandise (*ship* is a private action), and decides (namely, commits to itself) to inform the buyer that its request has been accepted and to update the stored amount of merchandise (*update-merchandise* is a private action).

```
(COMMIT
(?buyer REQUEST
            (DO now (ship ?buyer ?merchandise ?req-amnt ?price)))

(OR (AND (B (now (stored ?merchandise ?stored-amount)))
                (< ?stored-amount ?req-amnt))
            (AND (B (now (min-price ?merchandise ?min)))
                (⩽ ?price ?min)))

(myself (INFORM now ?buyer
            (refused ?merchandise ?req-amnt ?price)))
)
```

The second rule says that if the required amount of merchandise is not present in the warehouse or the price is too low, the seller agent decides to inform the buyer that its request has been refused.

```
(COMMIT
(?buyer REQUEST
            (DO now (ship ?buyer ?merchandise ?req-amnt ?price)))

(AND (B (now (stored ?merchandise ?stored-amount)))
                (⩾ ?stored-amount ?req-amnt)
                (B (now (max-price ?merchandise ?max)))
                (< ?price ?max)
                (B (now (min-price ?merchandise ?min)))
                (> ?price ?min))

(myself (DO now (eval-mean ?max ?price ?mean-price)))
```

```
(myself (REQUEST now ?buyer
            (eval-counter-proposal ?merchandise ?req-amnt ?mean-price)))
)
```

Finally, the third rule says that if the price can be negotiated and there is enough merchandise in the warehouse, the seller agent evaluates the price to propose to the buyer agent (*eval-mean* is a private action) and decides to send a counter-proposal to it. We are assuming that the buyer agent is able to perform an *eval-counter-proposal* action to evaluate the seller agent's proposal: the seller agent must know the exact action syntax if it wants that the buyer agent understands and satisfies its request.

5 Deontic logic

This introduction is based on the book *Deontic logic in Computer Science* (Meyer and Wieringa, 1993). Deontic logic is the logic to reason about ideal and actual behavior. From the 1950s, von Wright (von Wright, 1951), Castañeda (Castañeda, 1975), Alchourrón and Bulygin (Alchourrón and Bulygin, 1971) and others developed deontic logic as a modal logic with operators for permission, obligation and prohibition. Other operators are possible, such as formalizations of the system of concepts introduced by Hohfeld in 1913, containing operators for duty, right, power, liability, etc. (Hohfeld, 1913). Deontic logic has traditionally been used to analyze the structure of normative law and normative reasoning in law. Recently it has been realized that deontic logic can be of use outside the area of legal analysis and legal automation: it has a potential use in any area where we want to reason about ideal as well as actual behavior of systems. To give an idea of what deontic logic systems look like, we describe the OS Old System (von Wright, 1951) and the KD Standard System of Deontic Logic (Åqvist, 1984).

5.1 The OS System

The OS system is based on two deontic operators: **O**, meaning obligation, and **P** meaning permission. Let p be a proposition in the propositional calculus, then $\mathbf{O}p$ and $\mathbf{P}p$ are formulae in the OS deontic logic language.

The system consists of the following axioms and inference rule:

(OS0) All tautologies of Propositional Calculus
(OS1) $\mathbf{O}p \equiv \neg\mathbf{P}\neg p$
(OS2) $\mathbf{P}p \vee \mathbf{P}\neg p$
(OS3) $\mathbf{P}(p \vee q) \equiv \mathbf{P}p \vee \mathbf{P}q$
(OS4) $\frac{p \equiv q}{\mathbf{P}p \equiv \mathbf{P}q}$

Axiom (OS1) expresses that having an obligation to p is equivalent to not being permitted to not p; (OS2) states that either p is permitted or not p is; (OS3) says that a permission to p or q is equivalent to p being permitted or q being permitted; (OS4) asserts that if two assertions are equivalent, then permission for one implies permission for the other, and vice versa. Later, it was realized that the system OS is very close to a normal modal

logic, enabling a clear Kripke-style semantics using **O** as the basic necessity operator, at the expense of introducing the validity of

(O⊤) $\mathbf{O}(p \vee \neg p)$

stating the existence of an empty normative system, which von Wright rejected as an axiom.

5.2 The KD System

The KD System is a von Wright-type system including the **F** (forbidden) operator and consisting of the following axioms and rules.

(KD0) All tautologies of Propositional Calculus
(KD1) $\mathbf{O}(p \Rightarrow q) \Rightarrow (\mathbf{O}p \Rightarrow \mathbf{O}q)$
(KD2) $\mathbf{O}p \Rightarrow \mathbf{P}p$
(KD3) $\mathbf{P}p \equiv \neg\mathbf{O}\neg p$
(KD4) $\mathbf{F}p \equiv \neg\mathbf{P}p$
(KD5) Modus ponens: $\dfrac{p \qquad p \Rightarrow q}{q}$
(KD6) O-necessitation: $\dfrac{p}{\mathbf{O}p}$

Axiom (KD1) is the so called *K-axiom*; (KD2) is the *D-axiom*, stating that obligatory implies permitted; (KD3) states that permission is the dual of obligation and (KD4) says that forbidden is not permitted. (KD1) holds for any modal necessity operator. Essentially, it states that obligation is closed under implication. Whether this is desirable may be debatable, but it is a necessary consequence of the modal approach. Note furthermore that the O-necessitation rule (KD6), which is also part of the idea of viewing deontic logic as a normal modal logic, implies the axiom rejected by von Wright

(O⊤) $\mathbf{O}(p \vee \neg p)$

So, if we want to view deontic logic as a branch of Kripke-style modal logic, we have to commit ourselves to (O⊤).

As with other modal logics, the semantics of the standard system is based on the notion of a possible world. Given a Kripke model $\langle S, R, h\rangle$ and a world $s \in S$ we give the following semantics to the modal operators:

$$\begin{array}{lll}
\langle S,R,h\rangle \models_s \mathbf{O}p & \text{iff} & \text{for all } t \in S,\ s\,R\,t \text{ implies } \langle S,R,h\rangle \models_t p \\
\langle S,R,h\rangle \models_s \mathbf{P}p & \text{iff} & \text{exists } t \in S \text{ such that } s\,R\,t \ \wedge\ \langle S,R,h\rangle \models_t p \\
\langle S,R,h\rangle \models_s \mathbf{F}p & \text{iff} & \text{for all } t \in S,\ s\,R\,t \text{ implies } \langle S,R,h\rangle \not\models_t p
\end{array}$$

As already stated, the operator **O** is treated as the basic modal operator □: for $\mathbf{O}p$ being true in world s we have to check whether p holds in all the worlds reachable from s, as given by the relation R. This reflects the idea that something is obligated if it holds in all perfect (ideal) worlds (relative to the world where one is). This semantics is exactly the same semantics of □, as defined in Section 4. The other operators are more or less derived from **O**. The operator **P** is the dual of **O**: $\mathbf{P}p$ is true in the world s if there is some world reachable from s where p holds. Finally, something is forbidden in a world s if it does not hold in any world reachable from s.

5.3 The IMPACT agent language

We introduce the IMPACT agent programming language (Arisha *et al*., 1999; Eiter *et al*., 1999; Eiter and Subrahmanian, 1999; Eiter *et al*., 2000) as a relevant example of use of deontic logic to specify agents. To describe this language, we provide a set of definitions on top of which the language is based.

Agent Data Structures. All IMPACT agents are built "on top" of some existing body of code specified by the data types or data structures, $\mathscr{T}$, that the agent manipulates and by a set of functions, $\mathscr{F}$, that are callable by external programs. Such functions constitute the *application programmer interface* or API of the package on top of which the agent is being built.

Based on $\mathscr{T}$ and $\mathscr{F}$ supported by a package (a body of software code) $\mathscr{C}$, we may use a unified language to query the data structures. If $\mathtt{f} \in \mathscr{F}$ is an n-ary function defined in that package, and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are *terms* of appropriate types, then $\mathscr{C} : \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is a *code call*. This code call says "Execute function `f` as defined in package $\mathscr{C}$ on the stated list of arguments."

A *code call atom* is an expression `cca` of the form `in(t, cc)` or `notin(t, cc)`, where `t` is a term and `cc` is a code call. `in(t, cc)` evaluates to true (resp. false) in a given state if `t` is (resp. is not) among the values returned by calling `cc` in that state. The converse holds for `notin(t, cc)`. For example,

`in(⟨InOut, Sender, Receiver, Message, Time⟩, msgbox : getMessage(Sender))`

is true in a given state if the term `⟨InOut, Sender, Receiver, Message, Time⟩` is among the values returned by calling the `getMessage(Sender)` function provided by the `msgbox` package in that state.

A *code call condition* is a conjunction of code call atoms and *constraint atoms* of the form $\mathtt{t}_1$ *op* $\mathtt{t}_2$ where *op* is any of $=, \neq, <, \leqslant, >, \geqslant$ and $\mathtt{t}_1, \mathtt{t}_2$ are terms.

Each agent is also assumed to have access to a message box package identified by `msgbox`, together with some API function calls to access it (such as the `getMessage` function appearing in the code call atom above). Details of the message box in IMPACT may be found in Eiter *et al*. (1999).

At any given point in time, the actual set of objects in the data structures (and message box) managed by the agent constitutes the *state* of the agent. We shall identify a state $\mathscr{O}$ with the set of ground (namely, containing no variables) code calls which are true in it.

Actions. The agent can execute a set of *actions* $\alpha(X_1, \ldots, X_n)$. Such actions may include reading a message from the message box, responding to a message, executing a request, updating the agent data structures, etc. Even doing nothing may be an action. Expressions $\alpha(\vec{t})$, where $\vec{t}$ is a list of terms of appropriate types, are *action atoms*. Every action α has a precondition $Pre(\alpha)$ (which is a code call condition), a set of effects (given by an add list $Add(\alpha)$ and a delete list $Del(\alpha)$ of code call atoms) 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.

Notion of Concurrency. The agent has an associated body of code implementing a *notion of concurrency* **conc**$(AS, \mathscr{O})$. Intuitively, it takes a set of actions AS and the current agent state $\mathscr{O}$ as input, and returns a single action (which "combines" the input actions together)

as output. Various possible notions of concurrency are described in Eiter *et al*. (1999). For example, weak concurrent execution is defined as follows. Let AS be the set of actions in the current status set, evaluated according to the chosen semantics. Weakly concurrently executing actions in AS means that first all the deletions in the delete list of actions in AS are done in parallel and then all the insertions in the add list of actions in AS are. Even if some problems arise with this kind of concurrency, it has the advantage that deciding whether a set of actions is weakly-concurrent executable is polynomial (Theorem 3.1 of Eiter *et al*. (1999)).

Integrity Constraints. Each agent has a finite set $\mathcal{IC}$ of *integrity constraints* that the state $\mathcal{O}$ of the agent must satisfy (written $\mathcal{O} \models \mathcal{IC}$), 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. Informally, $\psi \Rightarrow \chi_a$ has the meaning of the universal statement "If ψ is true, then χ_a must be true." For simplicity, we omit here and in other places safety aspects (see Eiter *et al*. (1999) for details).

Agent Program. 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, must do, may not do, etc. Expressions $\mathbf{O}\alpha(\vec{t})$, $\mathbf{P}\alpha(\vec{t})$, $\mathbf{F}\alpha(\vec{t})$, $\mathbf{Do}\,\alpha(\vec{t})$, and $\mathbf{W}\alpha(\vec{t})$, where $\alpha(\vec{t})$ is an action atom, are called *action status atoms*. These action status atoms are read (respectively) as $\alpha(\vec{t})$ is *obligatory, permitted, forbidden, done*, and the obligation to do $\alpha(\vec{t})$ is *waived*. 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:

$$A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n \tag{5}$$

where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals.

5.3.1 Semantics

If an agent's behavior is defined by a program $\mathcal{P}$, the question that the agent must answer, over and over again is:

What is the set of all action status atoms of the form $\mathbf{Do}\,\alpha(\vec{t})$ that are true with respect to $\mathcal{P}$, the current state $\mathcal{O}$ and the set $\mathcal{IC}$ of underlying integrity constraints on agent states?

This set defines the actions the agent must take; Eiter *et al*. (1999) provide a series of successively more refined semantics for action programs that answer this question, that we discuss in a very succinct form.

Definition 1 (*Status Set*)
A *status set* is any set S of ground action status atoms over the values from the type domains of a software package $\mathcal{C}$.

Definition 2 (*Operator* $\boldsymbol{App}_{\mathcal{P},\mathcal{O}}(S)$)
Given a status set S, the operator $\mathbf{App}_{\mathcal{P},\mathcal{O}}(S)$ computes all action status atoms that may be inferred to be true by allowing the rules in $\mathcal{P}$ to fire exactly once. It is defined in the following way: let $\mathcal{P}$ be an agent program and $\mathcal{O}$ be an agent state. Then, $\mathbf{App}_{\mathcal{P},\mathcal{O}}(S) = \{Head(r\theta) \mid\ r \in \mathcal{P},\ R(r,\theta,S) \text{ is true on } \mathcal{O}\}$, where $Head(A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n) = A$ and the predicate $R(r,\theta,S)$ is true iff (1) $r\theta : A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n$ is a ground rule, (2)

$\mathcal{O} \models \chi$, (3) if $L_i = Op(\alpha)$ then $Op(\alpha) \in S$, and (4) if $L_i = \neg Op(\alpha)$ then $Op(\alpha) \notin S$, for all $i \in \{1, \ldots, n\}$.

Definition 3 (**A-Cl**(S))
A status set S is *deontically closed*, if for every ground action α, it is the case that ($DC1$) $\mathbf{O}\alpha \in S$ implies $\mathbf{P}\alpha \in S$. A status set S is *action closed*, if for every ground action α, it is the case that ($AC1$) $\mathbf{O}\alpha \in S$ implies $\mathbf{Do}\,\alpha \in S$, and ($AC2$) $\mathbf{Do}\,\alpha \in S$ implies $\mathbf{P}\alpha \in S$. It is easy to notice that status sets that are action closed are also deontically closed. For any status set S, we denote by **A-Cl**(S) the smallest set $S' \supseteq S$ such that S' is closed under ($AC1$) and ($AC2$), i.e. *action closed*.

Definition 4 (*Feasible Status Set*)
Let $\mathscr{P}$ be an agent program and let $\mathcal{O}$ be an agent state. Then, a status set S is a *feasible status set* for $\mathscr{P}$ on $\mathcal{O}$, if ($S1$)-($S4$) hold:

($S1$) $\mathbf{App}_{\mathscr{P},\mathcal{O}}(S) \subseteq S$;
($S2$) For any ground action α, the following holds: $\mathbf{O}\alpha \in S$ implies $\mathbf{W}\alpha \notin S$, and $\mathbf{P}\alpha \in S$ implies $\mathbf{F}\alpha \notin S$.
($S3$) $S = \mathbf{A\text{-}Cl}(S)$, i.e. S is action closed;
($S4$) The state $\mathcal{O}' = \mathbf{conc}(\mathbf{Do}\,(S), \mathcal{O})$ which results from $\mathcal{O}$ after executing (according to some execution strategy **conc**) the actions in $\{\alpha \mid \mathbf{Do}\,(\alpha) \in S\}$ satisfies the integrity constraints, i.e. $\mathcal{O}' \models \mathscr{IC}$.

Definition 5 (*Groundedness; Rational Status Set*)
A status set S is *grounded*, if no status set $S' \neq S$ exists such that $S' \subseteq S$ and S' satisfies conditions ($S1$)–($S3$) of a feasible status set. A status set S is a *rational status set*, if S is a feasible status set and S is grounded.

Definition 6 (*Reasonable Status Set*)
Let $\mathscr{P}$ be an agent program, let $\mathcal{O}$ be an agent state, and let S be a status set.

1. If $\mathscr{P}$ is positive, i.e. no negated action status atoms occur in it, then S is a *reasonable status set* for $\mathscr{P}$ on $\mathcal{O}$, iff S is a rational status set for $\mathscr{P}$ on $\mathcal{O}$.
2. The reduct of $\mathscr{P}$ w.r.t. S and $\mathcal{O}$, denoted by $red^S(\mathscr{P}, \mathcal{O})$, is the program which is obtained from the ground instances of the rules in $\mathscr{P}$ over $\mathcal{O}$ as follows.
   (a) Remove every rule r such that $Op(\alpha) \in S$ for some $\neg Op(\alpha)$ in the body of r;
   (b) remove all negative literals $\neg Op(\alpha)$ from the remaining rules.

Then S is a *reasonable status set* for $\mathscr{P}$ w.r.t. $\mathcal{O}$, if it is a reasonable status set of the program $red^S(\mathscr{P}, \mathcal{O})$ with respect to $\mathcal{O}$.

5.3.2 Implementation

The implementation of the IMPACT agent program consists of two major parts, both implemented in Java:

1. the IMPACT Agent Development Environment (IADE for short) which is used by the developer to build and compile agents, and

2. the run-time part that allows the agent to autonomously update its reasonable status set and execute actions as its state changes.

The IADE provides a network accessible interface through which an agent developer can specify the data types, functions, actions, integrity constraints, notion of concurrency and agent program associated with her/his agent; it also provides support for compilation and testing.

The runtime execution module runs as a background applet and performs the following steps: (i) monitoring of the agent's message box, (ii) execution of the algorithm for updating the reasonable status set and (iii) execution of the actions α such that **Do**α is in the updated reasonable status set.

5.3.3 Extensions

Many extensions to the IMPACT framework are discussed in the book (Subrahmanian *et al*., 2000) which analyzes:

- *meta agent programs* to reason about other agents based on the beliefs they hold;
- *temporal agent programs* to specify temporal aspects of actions and states;
- *probabilistic agent programs* to deal with uncertainty; and
- *secure agent programs* to provide agents with security mechanisms.

Agents able to recover from an integrity constraints violation and able to continue to process some requests while continuing to recover are discussed in Eiter *et al*. (2002). The integration of planning algorithms in the IMPACT framework is discussed in Dix *et al*. (2003).

5.3.4 Example

The IMPACT example appears to be more complicated than the other ones because we exemplify how it is possible to specify actions that require an access to external packages. These actions are defined in terms of their preconditions, add and delete list which involve the code call atoms that allow the real integration of external software. The ability of accessing real software makes IMPACT specifications more complex than the others we discuss in this paper but, clearly, also more powerful. We suppose that the IMPACT program for the seller agent accesses three software packages:

- an *oracle* database where information on the stored amount of merchandise and its minimum and maximum price is maintained in a *stored_merchandise* relation;
- a *msgbox* package that allows agents to exchange messages, as described in Section 3 of Eiter *et al*. (1999); in particular, it provides the *getMessage(Sender)* function which allows all tuples coming from *Sender* to be read and deleted from the message box of the receiving agent; and
- a mathematical package *math* providing mathematical functions.

- Initial state:

  The *stored_merchandise* relation, with schema $\langle$*name, amount, min, max*$\rangle$, initially contains the tuple $\langle$*orange, 1000, 1, 2*$\rangle$

- Actions:
  - *ship(Buyer, Merchandise, Req_amount)*

    Pre(ship(Buyer, Merchandise, Req_amount)) =
    in(Old_amount,
    oracle:select(stored_merchandise.amount, name, =, Merchandise)) ∧
    in(Difference, math:subtract(Old_amount, Req_amount)) ∧
    Difference ⩾ 0

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

    Del(ship(Buyer, Merchandise, Req_amount)) =
    in(Old_amount,
    oracle:select(stored_merchandise.amount, name, =, Merchandise))

    In order for the agent to ship merchandise, there must be enough merchandise available: the precondition of the action is true if the difference *Difference* between the stored amount of merchandise, *Old_amount*, and the required amount, *Req_amount*, is greater than or equal to zero. The effect of shipping is that the amount of available merchandise is updated by modifying the information in the *stored_merchandise* table: *stored_merchandise.amount* becomes equal to *Difference*, shared among the three equalities defining precondition, add list and delete list. *Add* and *Del* denote the desired modifications to the database through code calls. There is also a procedure, that we omit, that realizes this action. In practice this procedure would also issue an order to physically ship the merchandise.

  - *sendMessage(Sender, Receiver, Message)*
    This action accesses the *msgbox* package putting a tuple in the agent message box. The *msgbox* package underlying this action is assumed to ensure that tuples put in the message box are delivered to the receiver agent. The precondition of this action is empty (it is always possible to send a message), the add and delete lists consist of the updates to the receiver's mailbox.
    The notion of concurrency we adopt is *weak concurrent execution* introduced in Section 5.3.

- Integrity constraints:

  in(Min, oracle:select(stored_merchandise.min, name, =, Merchandise)) ∧
  in(Max, oracle:select(stored_merchandise.max, name, =, Merchandise)) ⇒
  0 < Min < Max
  This integrity constraint says that the minimum price allowed for any merchandise must be greater than zero and lower than the maximum price.

  in(Amount,
  oracle:select(stored_merchandise.amount, name, =, Merchandise)) ⇒
  Amount ⩾ 0

This integrity constraint says that any amount of merchandise must be greater or equal to zero.

$$in(\langle o, Sender, Receiver, accept(Merchandise, Req\_amount, Price), T\rangle,\\ msgbox{:}getMessage(Sender)) \wedge\\ in(\langle o, Sender, Receiver, refuse(Merchandise, Req\_amount, Price), T\rangle,\\ msgbox{:}getMessage(Sender) \Rightarrow\\ false$$

This integrity constraint says that an agent cannot both accept and refuse an offer (the *o* element in the tuple returned by the *msgbox:getMessage(Sender)* code call means that the message is an output message from *Sender* to *Receiver*; the last element of the tuple, *T*, is the time). Similar constraints could be added to enforce that an agent cannot both accept and negotiate an offer and that it cannot both refuse and negotiate.

$$in(Min, oracle{:}select(stored\_merchandise.min, name, =, Merchandise)) \wedge\\ in(\langle o, Sender, Receiver, accept(Merchandise, Req\_amount, Price), T\rangle,\\ msgbox{:}getMessage(Sender)) \wedge Price < Min \Rightarrow\\ false$$

This integrity constraint says that an agent cannot accept a proposal for a price lower than the minimum price allowed. Different from the previous ones, this constraint involves two different packages, the *oracle* one and the *msgbox* one.

Other integrity constraints could be added to ensure the consistency of data inside the same package or across different packages. Note that most of the above constraints are enforced by the agent program. The reason why they should be explicitly stated is that, in the case of legacy systems, the legacy system's existing interface and the agent both access and update the same data. Thus, the legacy interface may alter the agent's state in ways that the agent may find unacceptable. The violation of the integrity constraits prevents the agent from continuing its execution in an inconsistent state.

- Agent Program:

Do $sendMessage(Seller, Buyer, accept(Merchandise, Req\_amount, Price)) \leftarrow$
$in(\langle i, Buyer, Seller, contractProposal(Merchandise, Req\_amount, Price), T\rangle,$
$msgbox{:}getMessage(Seller)),$
$in(Max, oracle{:}select(stored\_merchandise.max, name, =, Merchandise)),$
$in(Amount, oracle{:}select(stored\_merchandise.amount, name, =, Merchandise)),$
$Price \geqslant Max, Amount \geqslant Req\_amount$

This rule says that if all the conditions for accepting a proposal are met, namely

1. the seller agent received a contractProposal from the buyer (the first element of the tuple in the first code call atom, *i*, says that the message is an input message),
2. there is enough merchandise in the warehouse and
3. the proposed price is greater than the *Max* value),

then the seller sends a message to the buyer, saying that it accepts the proposal.

O *ship(Buyer, Merchandise, Req_amount)* ←
  Do *sendMessage(Seller, Buyer, accept(Merchandise, Req_amount, Price))*

This rule says that if the seller agent accepts the buyer's proposal by sending a message to it, it is then obliged to ship the merchandise.

Do *sendMessage(Seller, Buyer, refuse(Merchandise, Req_amount, Price))* ←
  in(⟨i, Buyer, Seller, contractProposal(Merchandise, Req_amount, Price), T⟩,
    msgbox:getMessage(Seller)),
  in(Min, oracle:select(stored_merchandise.max, name, =, Merchandise)),
  $Price \leqslant Min$

If the price proposed by the buyer is below the *Min* threshold, then the seller agent refuses the proposal.

Do *sendMessage(Seller, Buyer, refuse(Merchandise, Req_amount, Price))* ←
  in(⟨i, Buyer, Seller, contractProposal(Merchandise, Req_amount, Price), T⟩,
    msgbox:getMessage(Seller)),
  in(Amount, oracle:select(stored_merchandise.amount, name, =, Merchandise)),
  $Amount < Req\_amount$

The proposal is refused if there is not enough merchandise available.

Do *sendMessage(Seller, Buyer,*
  contractProposal(Merchandise, Req_amount, Means)) ←
  in(⟨i, Buyer, Seller, contractProposal(Merchandise, Req_amount, Price), T⟩,
    msgbox:getMessage(Seller)),
  in(Max, oracle:select(stored_merchandise.max, name, =, Merchandise)),
  in(Min, oracle:select(stored_merchandise.min, name, =, Merchandise)),
  in(Amount, oracle:select(stored_merchandise.amount, name, =, Merchandise)),
  $Price > Min, Price < Max, Amount \geqslant Req\_amount$
  in(Means, math:evalMeans(Max, Price))

This rule manages the case the seller agent has to send a *contractProposal* back to the buyer, since the proposed price is between *Min* and *Max* and there is enough merchandise available.

6 Dynamic logic

Our introduction to dynamic logic is based on Section 8.2.5 of Weiss (1999). Dynamic logic can be thought of as the modal logic of action. Unlike traditional modal logics, the necessity and possibility operators of dynamic logic are based upon the kinds of actions available. As a consequence of this flexibility, dynamic logic has found use in a number of areas of Distributed Artificial Intelligence (DAI). We consider the propositional dynamic logic of regular programs, which is the most common variant. This logic has a sublanguage based on regular expressions for defining action expressions – these composite actions correspond to Algol-60 programs, hence the name of *regular programs*.

Regular programs and formulae of the dynamic logic language are defined by mutual induction. Let PA be a set of atomic action symbols. Regular programs are defined in the

following way:

- all atomic action symbols in PA are regular programs;
- if p and q are regular programs, then $p\,;q$ is a regular program meaning *doing p and q in sequence*;
- if p and q are regular programs, then $(p + q)$ is a regular program meaning *doing either p or q, whichever works*;
- if p is a regular program, then $p*$ is a regular program meaning *repeating zero or more (but finitely many) iterations of p*;
- if φ is a formula of the dynamic logic language, then $\varphi?$ is a regular program representing *the action of checking the truth value of formula* φ; it succeeds if φ is indeed found to be true.

$(p + q)$ is nondeterministic choice. This action might sound a little unintuitive since a nondeterministic program may not be physically executable, requiring arbitrary lookahead to infer which branch is really taken. From a logical viewpoint, however, the characterization of nondeterministic choice is clear. As far as $\varphi?$ is concerned, if φ is true, this action succeeds as a *noop*, i.e. without affecting the state of the world. If φ is false, it fails, and the branch of the action of which it is part terminates in failure – it is as if the branch did not exist.

Dynamic logic formulae are defined in the following way:

- all propositional formulae are dynamic logic formulae;
- if p is a regular program and φ is a dynamic logic formula, then $[p]\varphi$ is a dynamic logic formula which means that *whenever p terminates, it must do so in a state satisfying* φ;
- if p is a regular program and φ is a dynamic logic formula, then $\langle p\rangle\varphi$ is a dynamic logic formula which means that *it is possible to execute p and halt in a state satisfying* φ;
- if φ and ψ are dynamic logic formulae, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \Rightarrow \psi$ and $\neg\varphi$ are dynamic logic formulae.

Let S be a set of states (or worlds). Let h be an interpretation function

$$h : S \times PROP \rightarrow \{\text{true}, \text{false}\}$$

which says if a propositional formula belonging to *PROP* is true or false in a world belonging to S. Let $\sigma \subseteq S \times PA \times S$ be a transition relation.

The semantics of dynamic logic is given with respect to a model $\langle S, \sigma, h\rangle$ that includes a set S of states, a transition relation σ and an interpretation function h.

In order to provide the semantics of the language we first define a class of accessibility relations (β is a atomic action symbol from PA; p and q are regular programs; r, s and t, with subscripts when necessary, are members of S):

$s\ R_\beta\ t$	iff	$\sigma(s, \beta, t)$
$s\ R_{p;q}\ t$	iff	there exists r such that $s\ R_p\ r$ and $r\ R_q\ t$
$s\ R_{p+q}\ t$	iff	$s\ R_p\ t$ or $s\ R_q\ t$
$s\ R_{p*}\ t$	iff	there exists $s_0, \ldots, s_n$ such that $s = s_0$ and $t = s_n$ and for all $i, 0 \leqslant i < n, s_i\ R_p\ s_{i+1}$
$s\ R_{\varphi?}\ s$	iff	$\langle S, \sigma, h\rangle \models_s \varphi$

In the following equivalences p ranges over regular programs and φ and ψ are dynamic logic formulae.

If φ is a propositional formula, its semantics is given through the h interpretation function:

$$\langle S, \sigma, h\rangle \models_s \varphi \qquad \text{iff} \quad h(s, \varphi) = true$$

The semantics of $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \Rightarrow \psi$ and $\neg\varphi$ is given in the standard way:

$$\begin{array}{ll} \langle S, \sigma, h\rangle \models_s \varphi \vee \psi & \text{iff} \quad \langle S, \sigma, h\rangle \models_s \varphi \text{ or } \langle S, \sigma, h\rangle \models_s \psi \\ \langle S, \sigma, h\rangle \models_s \varphi \Rightarrow \psi & \text{iff} \quad \langle S, \sigma, h\rangle \models_s \varphi \text{ implies } \langle S, \sigma, h\rangle \models_s \psi \\ \text{etc} \ldots & \end{array}$$

The semantics of $\langle p\rangle\varphi$ and $[p]\varphi$ is given as:

$$\begin{array}{ll} \langle S, \sigma, h\rangle \models_s \langle p\rangle\varphi & \text{iff} \quad \text{there exists } t \text{ such that } s\ R_p\ t \text{ and } \langle S, \sigma, h\rangle \models_t \varphi \\ \langle S, \sigma, h\rangle \models_s [p]\varphi & \text{iff} \quad \text{for all } t,\ s\ R_p\ t \text{ implies } \langle S, \sigma, h\rangle \models_t \varphi \end{array}$$

The reader can refer to the survey by Kozen and Tiurzyn (1990) for additional details.

6.1 DyLOG

In a set of papers (Giordano *et al*., 1998; Giordano *et al*., 2000; Baldoni *et al*., 1997; Baldoni *et al*., 1998), Baldoni, Giordano, Martelli, Patti and Schwind describe an action language and its extension to deal with complex actions. In this section we provide a short description of the action language, taken from Baldoni *et al*. (2000), and we introduce an implementation. The implementation language is called DyLOG.

Primitive actions. In the action language each primitive action $a \in A$ is represented by a modality $[a]$. The meaning of the formula $[a]\alpha$, where α is an epistemic fluent, 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 in that state. They are called fluents because their value may change from state to state. The *simple action laws* are rules that allow to describe direct *action laws*, *precondition laws* and *causal laws*.

Action laws define the direct effect of primitive actions on a fluent and allow actions with conditional effects to be represented. They have the form

$$\Box(F_s \rightarrow [a]F)$$

where a is a primitive action name, F is a fluent, and F_s is a fluent conjunction, meaning that action a initiates F, when executed in a state where the *fluent precondition* F_s 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

$$\Box(F_s \rightarrow \langle a\rangle true)$$

meaning that when a fluent conjunction F_s 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

$$\Box(F_s \rightarrow F)$$

meaning that the fluent F holds if the fluent conjunction F_s holds too.

In the implementation language DyLOG the notation for the above constructs is the following: action laws have the form *a causes F if Fs*, precondition laws have the form *a possible_if Fs* and causal laws have the form *F if Fs*.

Procedures. *Procedures* are defined on the basis of primitive actions, test actions and other procedures. Test actions are needed for testing if some fluent holds in the current state and for expressing conditional procedures and are written as

$$F_s?$$

where F_s is a fluent conjunction.

A procedure p_0 is defined by means of a set of inclusion axiom schemas of the form

$$\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi \Rightarrow \langle p_0 \rangle \varphi$$

where φ stands for an arbitrary formula.

In DyLOG implementations a *procedure* is defined as a collection of *procedure clauses* of the form p_0 *isp* p_1 & ... & p_n ($n \geqslant 0$) where p_0 is the name of the procedure and p_i, $i = 1 \ldots n$ is either a primitive action, a test action (written $F_s?$), 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.

Planning. A *planning problem* amounts to determining, given an initial state and a goal F_s, if there is a possible execution of a procedure p leading to a state in which F_s holds. This can be formulated by the query

$$\langle p \rangle F_s$$

The execution of the above query returns as a side effect an answer which is an *execution trace* $a_1, a_2, \ldots, a_m$, i.e. a primitive action sequence from the initial state to the final one, which represents a *linear plan*.

To achieve this, DyLOG provides a metapredicate *plan(p, Fs, as)* where p is a procedure, *Fs* a goal and *as* a sequence of primitive actions. *plan* simulates the execution of the procedure p. If p includes sensing actions, the possible execution traces of p (represented by sequences *as* of primitive actions) are generated according to the possible outcomes of the sensing actions. When one (simulated) outcome leads to a final state where *Fs* is not satisfied, the interpreter backtracks and alternative outcome is simulated. Execution is separated from planning: a metapredicate *exe(as)* is provided to execute a plan.

Sensing. In general, it is not possible to assume that the value of each fluent in a state is known to an agent, and it is necessary to represent the fact that some fluents are unknown and to reason about the execution of actions on incomplete states. To represent explicitly the unknown value of some fluents, an *epistemic operator* **B** is introduced in the language, to represent the beliefs an agent has on the world. $\mathbf{B}f$ will mean that the fluent f is known to be true, $\mathbf{B}\neg f$ will mean that the fluent f is known to be false, and fluent f is undefined in the case both $\neg\mathbf{B}f$ and $\neg\mathbf{B}\neg f$ hold. In the following, $u(f)$ stands for $\neg\mathbf{B}f \wedge \neg\,\mathbf{B}\neg f$.

In DyLOG there is no explicit use of the operator **B** but the notation is extended with the test *u(f)?*. Thus each fluent can have one of the three values: true, false and unknown. An agent will be able to know the value of f by executing an action that senses f (*sensing* action). One expresses that an action s causes to know whether f holds by the declaration *s senses f*. By applying DyLOG's planning predicate *plan* to a procedure containing sensing actions a *conditional plan* is obtained. The branches of this plan correspond to the different outcomes of sensing actions.

6.1.1 Semantics

As discussed in Baldoni *et al.* (1998), the logical characterization of DyLOG can be 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.

Definition 7 (*Dynamic domain description*)
Given a set $\mathscr{A}$ of atomic world actions, a set $\mathscr{S}$ of sensing actions, and a set $\mathscr{P}$ of procedure names, let $\Pi_{\mathscr{A}}$ be a set of simple action laws for world actions, $\Pi_{\mathscr{S}}$ a set of axioms for sensing actions, and $\Pi_{\mathscr{P}}$ a set of inclusion axioms. A *dynamic domain description* is a pair (Π, S_0), where Π is the tuple $(\Pi_{\mathscr{A}}, \Pi_{\mathscr{S}}, \Pi_{\mathscr{P}})$ and S_0 is a consistent and complete set of epistemic literals representing the beliefs of the agent in the initial state.

Monotonic interpretation of a dynamic domain description. Given a dynamic domain description (Π, S_0), let us call $\mathscr{L}_{(\Pi,S_0)}$ the propositional modal logic on which (Π, S_0) is based. The action laws for primitive actions in $\Pi_{\mathscr{A}}$ and the initial beliefs in S_0 define a theory $\Sigma_{(\Pi,S_0)}$ in $\mathscr{L}_{(\Pi,S_0)}$. The axiomatization of $\mathscr{L}_{(\Pi,S_0)}$, called $\mathscr{S}_{(\Pi,S_0)}$, contains:

- all the axioms for normal modal operators
- $D(\mathbf{B})$, namely, for the belief modality **B** the axiom $\mathbf{B}p \Rightarrow \neg\mathbf{B}\neg p$ holds;
- $S4(\Box)$, namely, the three axioms $\Box p \Rightarrow p$, $\Box(p \Rightarrow q) \Rightarrow (\Box p \Rightarrow \Box q)$ and $\Box p \Rightarrow \Box\Box p$;
- $\Box\varphi \Rightarrow [a_i]\varphi$, one for each primitive action a_i in (Π, S_0);
- $\langle a + b\rangle\varphi \equiv \langle a\rangle\varphi \vee \langle b\rangle\varphi$, one for each formula φ;
- $\langle\psi?\rangle\varphi \equiv \psi \wedge \varphi$, one for each formula φ;
- $\langle a; b\rangle\varphi \equiv \langle a\rangle\langle b\rangle\varphi$, one for each formula φ;
- $\Pi_{\mathscr{P}}$;
- $\Pi_{\mathscr{S}}$.

The model theoretic semantics of the logic $\mathscr{L}_{(\Pi,S_0)}$ is given through a standard Kripke semantics with inclusion properties among the accessibility relations. More details can be found in Baldoni (1998).

Abductive semantics. The monotonic part of the language does not account for persistency. In order to deal with the frame problem, it is necessary to introduce a non-monotonic semantics for the language by making use of an abductive construction: abductive assumptions will be used to model persistency from one state to the following one, when a primitive action is performed. In particular, we will assume that a fluent expression F

persists through an action unless it is inconsistent to assume so, i.e. unless $\neg F$ holds after the action.

In defining the abductive semantics, the authors adopt (in a modal setting) the style of Eshghi and Kowalski's abductive semantics for negation as failure (Eshghi and Kowalski, 1989). They introduce the notation $\mathbf{M}\alpha$ to denote a new atomic proposition associated with α and a set of atomic propositions of the form $\mathbf{M}[a_1][a_2]\ldots[a_m]F$ and take them as being abducibles[2]. Their meaning is that the fluent expression F can be assumed to hold in the state obtained by executing primitive actions $a_1, a_2, \ldots, a_m$. Each abducible can be assumed to hold, provided it is consistent with the domain description (Π, S_0) and with other assumed abducibles.

More precisely, in order to deal with the frame problem, they add to the axiom system of $\mathscr{L}(\Pi, S_0)$ the *persistency axiom schema*

$$[a_1][a_2]\ldots[a_{m-1}]F \wedge \mathbf{M}[a_1][a_2]\ldots[a_{m-1}][a_m]F \Rightarrow [a_1][a_2]\ldots[a_{m-1}][a_m]F$$

where $a_1, a_2, \ldots, a_m (m > 0)$ are primitive actions, and F is a fluent expression. Its meaning is that, if F holds after action sequence $a_1, a_2, \ldots, a_{m-1}$, and F can be assumed to persist after action a_m (i.e. it is consistent to assume $\mathbf{M}[a_1][a_2]\ldots[a_m]F$), then we can conclude that F holds after performing the sequence of actions $a_1, a_2, \ldots, a_m$.

Besides the persistency action schema, the authors provide the notions of abductive solutions for a dynamic domain description and abductive solutions to a query.

6.1.2 Implementation

DyLOG is defined by a proof procedure which constructs a linear plan by making assumptions on the possible results of sensing actions. The goal directed proof procedure, based on negation as failure, allows a query to be proved from a given dynamic domain description. The proof procedure is sound and complete with respect to the Kripke semantics of the modal logics $\mathscr{L}(\Pi, S_0)$. An interpreter based on this proof procedure has been implemented in SICStus Prolog. This implementation allows to use DyLOG as a programming language for executing procedures which model the behavior of an agent, but also to reason about them, by extracting from them linear or conditional plans. Details on the implementation can be found in the ALICE Home Page (2000).

6.1.3 Extensions

In Baldoni *et al.* (2003a, 2003b), DyLOG agents are extended to represent beliefs of other agents in order to reason about conversations. They are also enriched with a communication kit including a primitive set of speech acts, a set of special "get message" actions and a set of conversation protocols.

[2] **M** is not a modality but just a notation adopted in analogy to default logic, where a justification $\mathbf{M}\alpha$ intuitively means "α is consistent".

6.1.4 Example

In the following example, the predicate *is* has its usual meaning as in Prolog programs: it evaluates the value of the expression at its right and checks if this value unifies with the term at its left.

- Functional fluents:

  functionalFluent(storing/2).
  functionalFluent(new_message/2).
  The amount of merchandise stored and the new incoming messages are facts which change during the agent's life. The number after the predicate's name is its arity.

- Unchangeable knowledge base (Prolog facts):

  min-price(orange, 1).
  max-price(orange, 2).
  The minimum and maximum prices for oranges do not change over time.

- Initial observations:

  obs(storing(orange, 1000)).
  Initially, there are 1000 oranges the seller agent can sell.

- Primitive actions:

  receive
  This action senses if a fluent *new_message(Sender, Message)* is present in the caller's mailbox. It is characterized by the following laws and routines:

  Precondition laws:
  receive possible_if true.
  It is always possible to wait for a new message to arrive.

  Sensing:
  receive senses new_message(Sender, Message).
  The *receive* action senses the value of the *new_message(Sender, Message)* functional fluent.

  Sensing routine:
  senses_routine(_, new_message, Sender, Message) :- ....
  A Prolog routine that we do not show here implements the sensing action by waiting for messages matching the couple *(Sender, Message)* in the caller's mailbox.

  send(Sender, Receiver, Message)
  This action puts the couple *(Sender, Message)* in the *Receiver*'s mailbox by modifying the state of the *new_message(Sender, Message)* functional fluent of the *Receiver*'s agent. For sake of conciseness, we avoid discussing all the details of this action.
  ship(Buyer, Merchandise, Req_Amnt, Price)
  This action ships the required merchandise to the *Buyer* agent. It is characterized by the following action laws and precondition laws:

Action laws:
ship(Buyer, Merchandise, Req_Amnt, Price)
causes storing(Merchandise, Amount)
if storing(Merchandise, Old_Amount) &
(Amount is Old_Amount - Req_Amnt).
Shipping some merchandise causes an update of the stored amount of that merchandise.

Precondition laws:
ship(Buyer, Merchandise, Req_Amnt)
possible_if storing(Merchandise, Old_Amount) &
(Old_Amount $\geqslant$ Req_Amnt) &
max-price(Merchandise, Max) & (Price $\geqslant$ Max).
Shipping merchandise is possible if there is enough merchandise left and if the price is higher than the maximum price established for that merchandise.

- 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(Merchandise, Req_Amnt, Price))? &
storing(Merchandise, Old_Amount)? &
(Old_Amount $\geqslant$ Req_Amnt) &
max-price(Merchandise, Max)? & (Price $\geqslant$ Max) &
ship(Buyer, Merchandise, Req_Amnt, Price) &
send(seller, Buyer, accept(Merchandise, Req_Amnt, Price))
If all conditions are met to ship the merchandise, then the merchandise is shipped and the seller sends a message to the *Buyer* in which it accepts the *Buyer*'s proposal.

manage_message isp
new_message(Buyer,
contractProposal(Merchandise, Req_Amnt, Price))? &
storing(Merchandise, Old_Amount)? &
(Old_Amount $<$ Req_Amnt) &
send(seller, Buyer, refuse(Merchandise, Req_Amnt, Price))
If there is not enough merchandise, the seller agent refuses to send the merchandise.

manage_message isp
new_message(Buyer,
contractProposal(Merchandise, Req_Amnt, Price))? &
min-price(Merchandise, Min)? & (Price ⩽ Min)
send(seller, Buyer, refuse(Merchandise, Req_Amnt, Price))
If the price is too low, the seller agent refuses to send the merchandise.

manage_message isp
new_message(Buyer,
contractProposal(Merchandise, Req_Amnt, Price))? &
storing(Merchandise, Old_Amount)? &
(Old_Amount ⩾ Req_Amnt) &
max-price(Merchandise, Max)? & (Price < Max) &
min-price(Merchandise, Min)? & (Price > Min) &
(New_Price is (Price + Max) / 2) &
send(seller, Buyer, contractProposal(Merchandise, Req_Amnt, New_Price))
In case the conditions are met to send a counter-proposal to the *Buyer* agent, the seller sends it with the price it is willing to accept.

7 Temporal logic

In this section we define a first-order temporal logic based on discrete, linear models with finite past and infinite future, called *FML* (Fisher, 1992). FML introduces two new connectives to classical logic, *until* ($\mathcal{U}$) and *since* ($\mathcal{S}$), together with a number of other operators definable in terms of $\mathcal{U}$ and $\mathcal{S}$. The intuitive meaning of a temporal logic formula $\varphi\ \mathcal{U}\psi$ is that ψ will become true at some future time point t and that in all states between and different from now and t, φ will be true. $\mathcal{S}$ is the analogous of $\mathcal{U}$ in the past.

Syntax of FML. Well-formed formulae of FML (WFF_f) are generated in the usual way as for classical logic, starting from a set $\mathcal{L}_p$ of predicate symbols, a set $\mathcal{L}_v$ of variable symbols, a set $\mathcal{L}_c$ of constant symbols, the quantifiers $\forall$ and $\exists$, and the set $\mathcal{L}_t$ of terms (constants and variables). The set WFF_f is defined by:

- If $t_1, \ldots, t_n$ are in $\mathcal{L}_t$ and p is a predicate symbol of arity n, then $p(t_1, \ldots, t_n)$ is in WFF_f.
- *true* and *false* are in WFF_f.
- If A and B are in WFF_f, then so are $\neg A$, $A \wedge B$, $A\ \mathcal{U} B$, $A\ \mathcal{S} B$, and (A).
- If A is in WFF_f and v is in $\mathcal{L}_v$, then $\exists v.A$ and $\forall v.A$ are both in WFF_f.

The other classical connectives are defined in terms of the ones given above, and several other useful temporal connectives are defined in terms of $\mathcal{U}$ and $\mathcal{S}$ (we follow the characterization provided in Finger *et al*. (1993) as well as the notation used there):

$\bigcirc\varphi$	φ is true in the next state	$[false\ \mathcal{U}\varphi]$
$\mathbf{O}\varphi$	the current state is not the initial state, and φ was true in the previous state	$[false\ \mathcal{S}\varphi]$
$\bullet\varphi$	if the current state is not the initial state, then φ was true in the previous state	$[\neg\ \mathbf{O}\neg\varphi]$

$\Diamond\varphi$	φ will be true in some future state	$[true\ \mathcal{U}\varphi]$
$\blacklozenge\varphi$	φ was true in some past state	$[true\ \mathcal{S}\varphi]$
$\Box\varphi$	φ will be true in all future states	$[\neg\ \Diamond\neg\varphi]$
$\blacksquare\varphi$	φ was true in all past states	$[\neg\blacklozenge\neg\varphi]$

Temporal formulae can be classified as follows. A *state-formula* is either a literal or a boolean combination of other state-formulae.
Strict future-time formulae are defined as follows:

> If A and B are either state or strict future-time formulae, then $A\ \mathcal{U}B$ is a strict future-time formula.
> If A and B are strict future-time formulae, then $\neg A$, $A \wedge B$, and (A) are strict future-time formulae.

Strict past-time formulae are defined as the past-time duals of strict future-time formulae.
Non-strict classes of formulae include state-formulae in their definition.

Semantics of FML. The models for FML formulae are given by a *structure* which consists of a sequence of states, together with an *assignment* of truth values to atomic sentences within states, a domain $\mathcal{D}$ which is assumed to be constant for every state, and mappings from elements of the language into denotations. More formally, a model is a tuple $\mathcal{M} = \langle \sigma, \mathcal{D}, h_c, h_p \rangle$ where σ is the ordered set of states $s_0, s_1, s_2, \ldots$, h_c is a map from the constants into $\mathcal{D}$, and h_p is a map from $\mathbf{N} \times \mathcal{L}_p$ into $\mathcal{D}^n \rightarrow \{$ true, false $\}$ (the first argument of h_p is the index i of the state s_i). Thus, for a particular state s, and a particular predicate p of arity n, $h(s, p)$ gives truth values to atoms constructed from n-tuples of elements of $\mathcal{D}$. A *variable* assignment h_v is a mapping from the variables into elements of $\mathcal{D}$. Given a variable and the valuation function h_c, a term assignment τ_{vh} is a mapping from terms into $\mathcal{D}$ defined in the usual way.

The semantics of FML is given by the $\models$ relation that gives 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.

$\langle \mathcal{M}, i, h_v \rangle \models true$		
$\langle \mathcal{M}, i, h_v \rangle \not\models false$		
$\langle \mathcal{M}, i, h_v \rangle \models p(x_1, \ldots, x_n)$	iff	$h_p(i, p)(\tau_{vh}(x_1), \ldots, \tau_{vh}(x_n)) = true$
$\langle \mathcal{M}, i, h_v \rangle \models \neg\varphi$	iff	$\langle \mathcal{M}, i, h_v \rangle \not\models \varphi$
$\langle \mathcal{M}, i, h_v \rangle \models \varphi \vee \psi$	iff	$\langle \mathcal{M}, i, h_v \rangle \models \varphi$ or $\langle \mathcal{M}, i, h_v \rangle \models \psi$
$\langle \mathcal{M}, i, h_v \rangle \models \varphi\ \mathcal{U}\psi$	iff	for some k such that $i < k$, $\langle \mathcal{M}, k, h_v \rangle \models \psi$ and for all j, if $i < j < k$ then $\langle \mathcal{M}, j, h_v \rangle \models \varphi$
$\langle \mathcal{M}, i, h_v \rangle \models \varphi\ \mathcal{S}\psi$	iff	for some k such that $0 \leqslant k < i$, $\langle \mathcal{M}, k, h_v \rangle \models \psi$ and for all j, if $k < j < i$ then $\langle \mathcal{M}, j, h_v \rangle \models \varphi$
$\langle \mathcal{M}, i, h_v \rangle \models \forall x \cdot \varphi$	iff	for all $d \in \mathcal{D}$, $\langle \mathcal{M}, i, h_v[d/x] \rangle \models \varphi$
$\langle \mathcal{M}, i, h_v \rangle \models \exists x \cdot \varphi$	iff	there exists $d \in \mathcal{D}$ such that $\langle \mathcal{M}, i, h_v[d/x] \rangle \models \varphi$

7.1 Concurrent METATEM

Concurrent METATEM (Fisher and Barringer, 1991; Fisher, 1993; Fisher and Wooldridge, 1993) is a programming language for distributed artificial intelligence based on FML.

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);
- a *computational engine*, which defines how the agent may act.

An agent interface consists of three components:

- a unique *agent identifier* which names the agent.
- a set of predicates defining what messages will be accepted by the agent – they are called *environment predicates*;
- a set of predicates defining messages that the agent may send – these are called *component predicates*.

Besides environment and component predicates, an agent has a set of *internal predicates* with no external effect.

The computational engine of an object is based on the METATEM paradigm of executable temporal logics. 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:

$$\text{antecedent about past} \Rightarrow \text{consequent about future}$$

The past-time antecedent is a temporal logic formula referring strictly to the past, whereas the future time consequent is a temporal logic formula referring either to the present or future. The intuitive interpretation of such a rule is *on the basis of the past, do the future*. The individual METATEM rules are given in the FML logic defined before. Since METATEM rules must respect the *past implies future* form, FML formulae defining agent rules must be transformed into this form. This is always possible as demonstrated in Barringer *et al*. (1990).

7.1.1 Semantics

METATEM semantics is the one defined for FML.

7.1.2 Implementation

Two implementations of the imperative future paradigm described in this section have been produced. The first is a prototype interpreter for propositional METATEM implemented in Scheme (Fisher, 1990). A more robust Prolog-based interpreter for a restricted first-order version of METATEM has been used as a transaction programming language for temporal databases (Finger *et al*., 1991).

7.1.3 Extensions

Two main directions have been followed in the attempt of extending Concurrent METATEM, the first one dealing with single agents and the second one dealing with MASs.

- Single Concurrent METATEM agents have been extended with deliberation and beliefs (Fisher, 1997) and with resource-bounded reasoning (Fisher and Ghidini, 1999).
- Compilation techniques for MASs specified in Concurrent METATEM are analyzed in Kellett and Fisher (1997a). Concurrent METATEM has been proposed as a coordination language in Kellett and Fisher (1997b). The definition of groups of agents in Concurrent METATEM is discussed in Fisher (1998) and Fisher and Kakoudakis (2000).

The research on single Concurrent METATEM agents converged with the research on Concurrent METATEM MASs in the paper by Fisher and Ghidini (2002) where "confidence" is added to both single and multiple agents. The development of teams of agents is discussed in Hirsch *et al.* (2002).

7.1.4 Example

The Concurrent METATEM program for the seller agent may be as follows:

- 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 with its arguments, not specified in the interface;
  – the messages that the seller agent is able to broadcast to the environment, including both communicative acts and actions on the environment, are *accept, refuse, contractProposal, ship* with their arguments.

- 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 (as usual, lowercase symbols are constants and uppercase ones are variables):

  $\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
  $\bigcirc$*[contractProposal(Buyer, seller, Merchandise, Req_Amnt, Price)* $\wedge$
  storing(Merchandise, Old_Amount) $\wedge$
  Old_Amount $\geqslant$ *Req_Amnt* $\wedge$
  max-price(Merchandise, Max) $\wedge$ *Price* $\geqslant$ *Max]* $\Rightarrow$
  [ship(Buyer, Merchandise, Req_Amnt, Price) $\wedge$
  accept(seller, Buyer, Merchandise, Req_Amnt, Price)]
  If there was a previous state where *Buyer* sent a *contractProposal* message to *seller*, and in that previous state all the conditions were met to accept the proposal, then accept the *Buyer*'s proposal and ship the required merchandise.

$\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
$\bigcirc$*[contractProposal(Buyer, seller, Merchandise, Req_Amnt, Price)* $\wedge$
storing(Merchandise, Old_Amount) $\wedge$
min-price(Merchandise, Min) $\wedge$
Old_Amount $<$ *Req_Amnt* $\vee$ *Price* $\leqslant$ *Min]* $\Rightarrow$
refuse(seller, Buyer, Merchandise, Req_Amnt, Price)

If there was a previous state where *Buyer* sent a *contractProposal* message to *seller*, and in that previous state the conditions were not met to accept the *Buyer*'s proposal, then send a *refuse* message to *Buyer*.

$\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
$\bigcirc$*[contractProposal(Buyer, seller, Merchandise, Req_Amnt, Price)* $\wedge$
storing(Merchandise, Old_Amount) $\wedge$
min-price(Merchandise, Min) $\wedge$
max-price(Merchandise, Max) $\wedge$
Old_Amount $\geqslant$ *Req_Amnt* $\wedge$
Price $>$ *Min* $\wedge$ *Price* $<$ *Max* $\wedge$
New_Price $=$ *(Max + Price) / 2]* $\Rightarrow$
contractProposal(seller, Buyer, Merchandise, Req_Amnt, New_Price)

If there was a previous state where *Buyer* sent a *contractProposal* message to *seller*, and in that previous state the conditions were met to send a *contractProposal* back to *Buyer*, then send a *contractProposal* message to *Buyer* with a new proposed price.

8 Linear logic

Linear logic (Girard, 1987) has been introduced as a resource-oriented refinement of classical logic. The idea behind linear logic is to constrain the number of times a given assumption (resource occurrence) can be used inside a deduction for a given goal formula. This resource management, together with the possibility of naturally modeling the notion of state, makes linear logic an appealing formalism to reason about concurrent and dynamically changing systems.

Linear logic extends usual logic with new connectives:

- *Exponentials*: "!" (*of course*) and "?" (*why not?*) express the capability of an action of being iterated, i.e. the absence of any reaction. $!A$ means infinite amount of resource A.
- *Linear implication*: $\multimap$ (*lolli*) is used for causal implication. The relationship between linear implication and intuitionistic implication "$\Rightarrow$" is $A \Rightarrow B \equiv (!A) \multimap B$
- *Conjunctions*: $\otimes$ (*times*) and & (*with*) correspond to radically different uses of the word "and". Both conjunctions express the availability of two actions; but in the case of $\otimes$, both actions will be done, whereas in the case of & only one of them will be performed (we shall decide which one). Given an action of type $A \multimap B$ and an action of type $A \multimap C$ there will be no way of forming an action of type $A \multimap B \otimes C$, since once resource A has been consumed for deriving B, for example, it is not available for deriving C. However, there will be an action $A \multimap B \& C$. In order to perform this

Table 1. *A one-sided, dyadic proof system for linear logic*

$\dfrac{}{\vdash \Theta : F, F^{\perp}}\ id$	$\dfrac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Delta, F^{\perp}}{\vdash \Theta : \Gamma, \Delta}\ cut$	$\dfrac{\vdash \Theta, F : \Gamma, F}{\vdash \Theta, F : \Gamma}\ abs$
$\dfrac{\vdash \Theta : \Gamma}{\vdash \Theta : \Gamma, \perp}\ \perp$	$\dfrac{\vdash \Theta : \Gamma, F, G}{\vdash \Theta : \Gamma, F \parr G}\ \parr$	$\dfrac{\vdash \Theta, F : \Gamma}{\vdash \Theta : \Gamma, ?F}\ ?$
$\dfrac{}{\vdash \Theta : \mathbf{1}}\ \mathbf{1}$	$\dfrac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Delta, G}{\vdash \Theta : \Gamma, \Delta, F \otimes G}\ \otimes$	$\dfrac{\vdash \Theta : F}{\vdash \Theta : !F}\ !$
$\dfrac{}{\vdash \Theta : \Gamma, \top}\ \top$	$\dfrac{\vdash \Theta : \Gamma, F \qquad \vdash \Theta : \Gamma, G}{\vdash \Theta : \Gamma, F \& G}\ \&$	$\dfrac{\vdash \Theta : \Gamma, F[c/x]}{\vdash \Theta : \Gamma, \forall x.F}\ \forall$
$\dfrac{\vdash \Theta : \Gamma, F}{\vdash \Theta : \Gamma, F \oplus G}\ \oplus_l$	$\dfrac{\vdash \Theta : \Gamma, G}{\vdash \Theta : \Gamma, F \oplus G}\ \oplus_r$	$\dfrac{\vdash \Theta : \Gamma, F[t/x]}{\vdash \Theta : \Gamma, \exists x.F}\ \exists$

action we have to first choose which among the two possible actions we want to perform, and then do the one selected.

- *Disjunctions*: there are two disjunctions in linear logic, $\oplus$ (*plus*), which is the dual of &, and $\parr$ (*par*), which is the dual of $\otimes$. $\oplus$ expresses the choice of one action between two possible types. $\parr$ expresses a dependency between two types of actions and can be used to model concurrency.
- *Linear negation*: $(\cdot)^{\perp}$ (*nil*) expresses linear negation. Since linear implication will eventually be rewritten as $A^{\perp} \parr B$, *nil* is the only negative operation of logic. Linear negation expresses a *duality*

$$\textit{action of type } A = \textit{reaction of type } A^{\perp}$$

- *Neutral elements*: there are four neutral elements: $\mathbf{1}$ (w.r.t. $\otimes$), $\perp$ (w.r.t. $\parr$), $\top$ (w.r.t. &) and $\mathbf{0}$ (w.r.t. $\oplus$).

Semantics. In Table 1 we provide the semantics of full linear logic by means of a proof system. Sequents assume the one-sided, dyadic form $\vdash \Theta : \Gamma$. Θ and Γ are multisets of formulae. Θ is the so-called *unbounded* part, while Γ is the *bounded* one. In other words, formulae in Θ must be implicitly considered as exponentiated (i.e. preceded by ?) and thus can be reused any number of times, while formulae in Γ must be used exactly once.

We opted for defining the semantics of linear logic by means of a proof system both because understanding the proof rules requires less background than understanding an abstract semantics and for consistency with the style used for the semantics of $\mathscr{E}_{hhf}$. Other semantics have been defined for linear logic: a complete description of *phase semantics* and *coherent semantics* is given in Girard (1987) while *game semantics* is dealt with in Blass (1992).

8.1 $\mathscr{E}_{hhf}$

The language $\mathscr{E}_{hhf}$ (Delzanno, 1997; Delzanno and Martelli, 2001) is an executable specification language for modeling concurrent and resource sensitive systems, based on the general purpose specification logical language Forum (Miller, 1996). $\mathscr{E}_{hhf}$ 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. $\mathscr{E}_{hhf}$ uses a subset of linear logic connectives and a restricted class of formulae as defined later. An $\mathscr{E}_{hhf}$-program P is a collection of multi-conclusion clauses of the form:

$$A_1 \parr \ldots \parr A_n \circ\!\!- Goal,$$

where the A_i are atomic formulae, the linear disjunction $A_1 \parr \ldots \parr A_n$ corresponds to the head of the clause and *Goal* is its body. Furthermore, $A \circ\!\!- B$ is a linear implication. Execution of clauses of this kind concurrently *consumes* the resources (formulae) they need in order to be applied in a resolution step.

Given a multiset of atomic formulae (the state of the computation) Ω_0, a resolution step $\Omega_0 \rightarrow \Omega_1$ can be performed by applying an instance $A_1 \parr \ldots \parr A_n \circ\!\!- G$ of a clause in the program P, whenever the multiset Θ consisting of the atoms $A_1, \ldots, A_n$ is contained in Ω_0. Ω_1 is then obtained by removing Θ from Ω_0 and by adding G to the resulting multiset. In the $\mathscr{E}_{hhf}$ interpreter, instantiation is replaced by unification. At this point, since G may be a complex formula, the search rules (i.e. the logical rules of the connectives occurring in G) must be exhaustively applied in order to proceed. Such a derivation corresponds to a specific branch of the proof tree of a multiset Ω. Ω represents the current global state, whereas P describes a set of clauses that can be triggered at any point during a computation.

$\mathscr{E}_{hhf}$ provides a way to "guard" the application of a given clause. In the extended type of clauses

$$G_1 \,\&\, \ldots \,\&\, G_m \Rightarrow (A_1 \parr \ldots \parr A_n \circ\!\!- Goal),$$

the goal-formulae G_i are *conditions* that must be solved in order for the clause to be triggered.

New components can be added to the current state by using goal-formulae of the form $G_1 \parr G_2$. In fact, the goal $G_1 \parr G_2, \Delta$ simply reduces to G_1, G_2, Δ. Conditions over the current state can be tested by using goal-formulae of the form $G_1 \& G_2$. In fact, the goal $G_1 \& G_2, \Delta$ reduces to G_1, Δ and G_2, Δ. Thus, one of the two copies of the state can be consumed to verify a contextual condition. Universal quantification in a goal-formula $\forall x.G$ can be used to create a new identifier t which must be local to the derivation tree of the subgoal $G[t/x]$. Finally, the constant $\top$ succeeds in any context and the constant $\bot$ is simply removed from the current goal.

8.1.1 Semantics

The $\mathscr{E}_{hhf}$ operational semantics is given by means of a set of rules describing the way sequents can be rewritten. See Delzanno and Martelli (2001) for all details and for the results of correctness and completeness w.r.t. linear logic. According to the *proof as computation interpretation* of linear logic, sequents represent the state of a computation.

Sequents assume the following form (simplified for the sake of presentation):

$$\Gamma ; \Delta \rightarrow \Omega,$$

where Γ and Δ are multisets of $\mathscr{D}$-formulae (respectively, the unbounded and bounded context), and Ω is a multiset of $\mathscr{G}$-formulae which contains the concurrent resources present in the state. The class of formulae is restricted to two main classes, $\mathscr{D}$- and $\mathscr{G}$-formulae ($\mathscr{D}$ stands for *definite* clauses and $\mathscr{G}$ for *goals*):

$$\begin{aligned}
\mathscr{D} &::= \mathscr{D} \,\&\, \mathscr{D} \mid \forall x \cdot \mathscr{D} \mid \mathscr{H} \circ\!\!- \mathscr{G} \mid \mathscr{D} \Leftarrow G \mid \mathscr{H} \\
\mathscr{G} &::= \mathscr{G} \,\&\, \mathscr{G} \mid \mathscr{G} \parr \mathscr{G} \mid \forall x \cdot \mathscr{G} \mid \mathscr{D} \multimap \mathscr{G} \mid \mathscr{D} \Rightarrow \mathscr{G} \mid A \mid \bot \mid \top \\
\mathscr{H} &::= \mathscr{H} \parr \mathscr{H} \mid A_r
\end{aligned}$$

A represents a generic atomic formula, whereas A_r is a *rigid* atomic formula, i.e. whose top-level functor symbol is a constant.

The rules of $\mathscr{E}_{hhf}$ are divided into *right rules* (or *search rules*) and *left rules*. Right rules are used to simplify goals in Ω until they become atomic formulae. They define the behavior of the various connectives: $\top$ is used to manage termination, $\bot$ to encode a null statement, & to split a computation into two branches which share the same resources, $\forall$ to encode a notion of *hiding*, $\parr$ to represent concurrent execution, $\Rightarrow$ and $\multimap$ to augment the resource context (respectively, the unbounded and the bounded context).

Left rules define backchaining over clauses built with the connectives $\Leftarrow$ and $\circ\!\!-$. As described above, the rule for $\circ\!\!-$ is similar to Prolog rewriting, except that multiple-headed clauses are supported; besides, a clause can be reusable or not depending on which context it appears in. The rule for $\Leftarrow$ allows to depart an independent branch in an empty context (this is often useful to verify side conditions or make auxiliary operations).

8.1.2 Implementation

A working interpreter for $\mathscr{E}_{hhf}$ has been developed by Bozzano in Lambda Prolog, a language originally developed by Miller and Nadathur, which offers support for higher-order abstract syntax, a new and increasingly popular way to view the structure of objects such as formulae and programs.

The code of the $\mathscr{E}_{hhf}$ interpreter can be downloaded from (Ehhf FTP Area, 1998), where an example implementing the specification described in (Bozzano *et al.*, 1999a) is also downloadable.

8.1.3 Extensions

In the MAS context, $\mathscr{E}_{hhf}$ has been used to specify an architecture based on the BDI (*Belief, Desires, Intentions* (Rao and Georgeff, 1995)) approach (Bozzano *et al.*, 1999b) and to verify the correctness of a MAS where agents were specified by means of event–condition–action rules (Bozzano *et al.*, 1999a).

$\mathscr{E}_{hhf}$ has also been used to model object-oriented and deductive databases (Bozzano *et al.*, 1997) and object calculi (Bugliesi *et al.*, 2000).

8.1.4 Example

The $\mathscr{E}_{hhf}$ program for the seller agent may be as follows:

- Seller's initial facts:

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

  We assume that every agent has a mailbox which all the agents in the system can update by calling a *send* predicate. The mailbox of the seller agent is initially empty (we are using Prolog syntax for lists).

- Seller's life cycle:

  $\forall$ *Message, OtherMessages.*
  seller-mailbox([Message|OtherMessages]) $\otimes$
  seller-cycle $\circ\!\!-$
  manage(Message) $\otimes$
  seller-mailbox(OtherMessages) $\otimes$
  seller-cycle.

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

- Seller's rules for managing messages:

  $\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
  Old_Amount $\geqslant$ *Req_Amnt* &
  difference(Old_Amount, Req_Amnt, Remaining_Amnt) &
  max-price(Merchandise, Max) & *Price* $\geqslant$ *Max* $\Rightarrow$
  manage(contractProposal(Buyer, Merchandise, Req_Amnt, Price)) $\otimes$
  storing(Merchandise, Old_Amount) $\circ\!\!-$
  storing(Merchandise, Remaining_Amount) $\otimes$
  ship(Buyer, Merchandise, Req_Amnt, Price) $\otimes$
  send(Buyer, accept(seller, Merchandise, Req_Amnt, Price)).

  The goals before the $\Rightarrow$ connective are not consumed by the execution of the rule: they are used to evaluate values (*difference(Old_Amount, Req_Amnt, Remaining_Amnt)*), to compare values (*Old_Amount* $\geqslant$ *Req_Amnt* and *Price* $\geqslant$ *Max*) and to get the value of variables appearing in facts that are not changed by the rule (*max-price(Merchandise, Max)*). In this case, they succeed if the conditions for shipping merchandise are met. The goals *storing(Merchandise, Old_Amount)* and *manage(contractProposal(Buyer, Merchandise, Req_Amnt, Price))* are consumed; they are rewritten in *storing(Merchandise, Remaining_Amount)* (the information about stored merchandise is updated), *ship(Buyer, Merchandise, Req_Amnt, Price)* (the required amount of merchandise is shipped) and

send(Buyer, accept(seller, Merchandise, Req_Amnt, Price) (the message for informing *Buyer* that its proposal has been accepted is sent). The *ship* predicate will be defined by some rules that we do not describe here.

$\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
min-price(Merchandise, Min) & $Price \leqslant Min \Rightarrow$
manage(contractProposal(Buyer, Merchandise, Req_Amnt, Price)) $\circ\!-$
send(Buyer, refuse(seller, Merchandise, Req_Amnt, Price).

If the proposed price is too low (*min-price(Merchandise, Min)* & $Price \leqslant Min$) the *Buyer*'s proposal is refused.

$\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
storing(Merchandise, Old_Amount) & $Old\_Amount < Req\_Amnt \Rightarrow$
manage(contractProposal(Buyer, Merchandise, Req_Amnt, Price)) $\circ\!-$
send(Buyer, refuse(seller, Merchandise, Req_Amnt, Price)).

If there is not enough merchandise stored, the *Buyer*'s proposal is refused.

$\forall$ *Buyer, Merchandise, Req_Amnt, Price.*
$Old\_Amount \geqslant Req\_Amnt$ &
min-price(Merchandise, Min) & $Price > Min$ &
max-price(Merchandise, Max) & $Price < Max$ &
eval-means(Max, Price, Means) $\Rightarrow$
manage(contractProposal(Buyer, Merchandise, Req_Amnt, Price)) $\circ\!-$
send(Buyer,
contractProposal(seller, Merchandise, Req_Amnt, Means)).

If there is enough merchandise and the price proposed by *Buyer* is between the minimum and maximum prices established by the seller, the means of *Price* and *Max* is evaluated (*eval-means(Max, Price, MeanPrice)*) and a *contractProposal* with this new price is sent to *Buyer*.

9 A comparison among the specification languages

In this section we compare the agent specification languages introduced so far along twelve dimensions whose choice is mainly driven by (Juan *et al*., 2003a). Although it is difficult to assess if the following twelve dimensions are all and the only ones relevant for characterizing an agent programming language, we think that they represent a reasonable choice.

In Section 9.1 we introduce the twelve dimensions. For each one we explain why it is relevant for characterizing an agent programming language. We also formulate some questions whose answers, given in Section 9.2, help in understanding how each of the six languages analyzed in this paper supports the given dimension.

9.1 Comparison dimensions

This paper is mainly concerned with the prototyping stage rather than with the development of a final application. For this reason we avoid discussing all those technical details which are not necessary for modeling, verifying and prototyping a MAS, such as efficiency,

support for mobility and physical distribution, support for integration of external packages. Indeed, we concentrate on dimensions related with the basic definition of an agent quoted in the introduction (Jennings *et al*., 1998) (dimensions 2, 3, 4, 5), on dimensions related with the agent representation and management of knowledge (dimension 6), on dimensions related with the ability of a set of agents to form a MAS (dimensions 7, 8, 9), and on dimensions which, although not peculiar of an agent programming language, are particularly important for the correct development of agents and a MAS (dimensions 10, 11, 12).

1. *Purpose of use.* Understanding in which engineering/development stage the language proves useful is necessary to adopt the right language at the right time.
   - Is the language suitable for running autonomous agents in a real environment?
   - Is the language suitable for MAS prototyping?
   - Is the language suitable for verifying properties of the implemented MAS?
2. *Time.* Agents must both 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.
   - Is time dealt with explicitly in the language?
   - Are there operators for defining complex timed expressions?
3. *Sensing.* One of the characterizing features of an agent is its ability to sense and perceive the surrounding environment.
   - Does the language provide constructs for sensing actions (namely, actions which sense the environment)?
4. *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).
   - Does the language allow the modeling of concurrent actions within the same agent?
   - Does it support concurrency among executing agents?
5. *Nondeterminism.* The evolution of a MAS consists of a nondeterministic succession of events.
   - Does the language support nondeterminism?
6. *Agent knowledge.* The predominant agent model attributes human-like attitudes to agents. The agent knowledge is often characterized by beliefs, desires and intentions (Rao and Georgeff, 1995). Often, human beings are required to reason in presence of incomplete and uncertain knowledge.
   - Does the language support a BDI-style architecture?
   - Does the language support incomplete agent knowledge?
   - Does it support uncertainty?
7. *Communication.* Agents must be social, namely, they must be able to communicate either with other agents and with human beings.
   - Are communication primitives provided by the language?

- Is it necessary for an agent to know details of another agent's implementation in order to communicate with it, or does communication take place on a more abstract level?
- Is the programming language tied to some specific agent communication language?

8. *Team working*. The ability to form team is becoming more and more important in the intelligent agents research area as witnessed by the increasing number of researchers which address this specific topic[3]. Building a team may involve coordination/negotiation protocols.
   - Is the language suitable for defining and programming teams?
   - Is the language suitable for expressing coordination/negotiation protocols?
9. *Heterogeneity and knowledge sharing*. In many real systems agents are heterogeneous since they were developed by different organizations with different (sometimes opposite) purposes in mind.
   - Which are the necessary conditions that agents must respect to interact?
   - Do agents need to share the same ontology?
   - Are agents able to cope with the heterogeneity of information?
10. *Programming style*. The language programming style may be more or less suitable for implementing a given reasoning mechanism or a given agent architecture.
    - Does the language support goal-directed reasoning, forward reasoning, reactiveness?
    - Does the agent programming language require to stick to a fixed agent model or does it leave the choice to the programmer?
11. *Modularity*. Agent programs are typically very complex and a developer would benefit from structuring them by defining modules, macros and procedures.
    - Does the language provide constructs for defining modules, macros and/or procedures?
12. *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.
    - Is a formal semantics of the language defined?
    - Are there results explaining the link between system execution and formal semantics?

9.2 The six languages compared along the twelve dimensions

Purpose of use. ConGolog allows the design of flexible controllers for agents living in complex scenarios. Its extension IndiGolog provides a practical framework for real robots

[3] For example, during the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS) which took place in Melbourne in July 2003 an entire session was devoted to team working, with four papers presented; a large number of documents on team work is published by the TEAMCORE Research Group at the University of Southern California (The TEAMCORE Research Group Home Page, 2003).

that must sense the environment and react to changes occurring in it, and Legolog is an agent architecture for running IndiGolog high-level programs in Lego MINDSTORM robots. CASL (Shapiro *et al.*, 2002)) is an environment based on ConGolog which provides a verification environment.

AGENT-0 is suitable for modeling agents and MAS. We are not aware of papers on the suitability of AGENT-0 or its extensions for verifying MAS specifications or implementing real agent systems.

IMPACT's main purpose is to allow the integration of heterogeneous information sources and software packages. It has been used to develop real applications ranging from combat information management where IMPACT was used to provide yellow pages matchmaking services to aerospace applications where IMPACT technology has led to the development of a multiagent solution to the "controlled flight into terrain" problem. The IADE environment provides support for monitoring the MAS evolution.

DyLOG is suitable for building agents acting, interacting and planning in dynamic environments. A web agent system called WLog has been developed using DyLOG to demonstrate DyLOG's potential in developing adaptative web applications as software agents.

In Fisher (1994) a range of sample applications of Concurrent METATEM utilizing both the core features of the language and some of its extensions are discussed. They include bidding, problem solving, process control, fault tolerance. Concurrent METATEM has the potential of specifying and verifying applications in all of the areas above (Fisher and Wooldridge, 1997), but it is not suitable for the development of real systems.

$\mathscr{E}_{hhf}$ can be used for MAS modeling and verification, as discussed in (Bozzano *et al.*, 1999a). It is not suitable for running autonomous agents in a real environment.

Time. In ConGolog time instants correspond directly with situations: s_0 is the agent's situation at time 0, $do([a_1, \ldots, a_n], s_0)$ is the agent's situation at time n. We can think of a succession of situations as a discrete time line. Most temporal modalities as found in temporal logics can be expressed in situation calculus using quantification over situations.

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 specify the time grain 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, sometime in the past, sometime in the future, always in the past, always in the future*") which allow the definition of 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. DyLOG is the only language which provides an explicit construct for defining actions which sense the value of a fluent. However all the languages allow perception of 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. In our running example, all the agents maintain the information about the stored amount of oranges locally in their knowledge bases. In practice this information should be obtained

by physically sensing the environment (the warehouse, in this case), since nothing ensures that the agent's information is consistent with the environment state.

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 accessing external software packages.

We also note that, although the ConGolog language does not support sensing primitives, IndiGolog does, and that sensing in the situation calculus is discussed by Reiter (2001).

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 $\mathscr{E}_{hhf}$, 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 a *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 among agents cannot be explicitly specified.

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.

Both DyLOG and AGENT-0 do not support concurrency at the language level.

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 are not unique, thus introducing nondeterminism in the agent's behavior.

In DyLOG and $\mathscr{E}_{hhf}$ nondeterminism is introduced, as in usual logic programming settings, by the presence of more procedures (rules, in $\mathscr{E}_{hhf}$) defining the same predicate.

The main source of nondeterminism in Concurrent METATEM is due to nondeterministic temporal operators such as "sometime in the past", "sometime in the future", which do not identify a specific point in time, but may be verified in a range of time points.

AGENT-0 does not seem to support any kind of nondeterministic behavior.

Agent knowledge. A ConGolog model can include the specification of the agents' mental states, i.e. what knowledge and goals they have, specified in a purely declarative way (Shapiro *et al.*, 1998). With respect to incomplete knowledge,

> ConGolog can accommodate incompletely specified models, both in the sense that the initial state of the system is not completely specified, and in the sense that the processes involved are nondeterministic and may evolve in any number of ways (Lespérance and Shapiro, 1999).

AGENT-0 allows for expressing beliefs, capabilities, commitments, obligations. It does not allow the representation of intentions, goals and desires and it does not support reasoning mechanisms in presence of incomplete knowledge or uncertainty. PLACA adds intentions and plans to the data structures provided by AGENT-0.

Beliefs of IMPACT agents consist of the values returned by the packages accessed by the agent by means of the code call mechanism. No intentions and desires are ascribed to IMPACT agents: an IMPACT agent is characterized by its obligations, permissions, prohibitions. Two extensions of basic IMPACT agent programs, namely probabilistic and meta agent programs, can deal with uncertainty and beliefs about other agents beliefs, respectively.

Beliefs of DyLOG agents are represented by the values of the functional fluents characterizing the agent's knowledge. These values range over true, false and unknown. The "unknown" value allows DyLOG agents to reason in presence of incomplete knowledge, as discussed by Baldoni *et al*. (2001). Agents can perform hypothetical reasoning on possible sequences of actions by exploring different alternatives. Recent extensions allow DyLOG agents to represent beliefs of other agents to reason about conversation protocols (Baldoni *et al*., 2003a,b).

The beliefs of Concurrent METATEM agents in a given time point consist of the set of predicates true in that time point. Adding deliberation and explicit beliefs to Concurrent METATEM agents is discussed in Fisher (1997) and the extension of Concurrent METATEM agents with confidence is dealt with in Fisher and Ghidini (2002).

In $\mathscr{E}_{hhf}$ beliefs are represented by true facts. Goals are neither explicitly represented nor maintained in persistent data structures during the agent execution: they are managed in the usual way in a logic programming setting. $\mathscr{E}_{hhf}$ does not provide language constructs for representing desires, intentions and other mental attitudes, but it may be adopted to model a BDI architecture (Bozzano *et al*., 1999b).

Communication. The specification of communicative multiagent systems in ConGolog is discussed in Shapiro *et al*. (1998). A meeting scheduler multiagent system example is used to show in practice the proposed approach.

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. Communication in AGENT-0 is quite rigid since, for agent A to request an action to agent B it is necessary to know the exact syntax of the requested action. The receiver agent has no means to understand the content of a request and perform 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 recent agent communication languages, such as KQML (Mayfield *et al*., 1995) and FIPA ACL (Foundation for Intelligent Physical Agents, 2002) have partially addressed. The integration of AGENT-0 and KQML proposed in Davies and Edwards (1994) aims at making communication management in AGENT-0 more flexible.

The same limitation affecting AGENT-0 also affects Concurrent METATEM: every agent has a communicative interface which the other agents in the system must know in order to exchange information. Despite this limitation, Concurrent METATEM has been proposed both as a coordination language (Kellett and Fisher, 1997b) and as a language

for forming groups of agents where agents have the ability to broadcast messages to the members of a group (Fisher and Kakoudakis, 2000; Fisher, 1998).

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.

In Baldoni *et al*. (2003b) DyLOG agents are enriched with a communication kit including a primitive set of speech acts, a set of special "get message" actions and a set of conversation protocols. Exchanged messages are based on FIPA ACL.

An agent behavior driven by the reception of a message and the verification of a condition on the current state can be easily modeled in $\mathscr{E}_{hhf}$. In Bozzano *et al*. (1999a) the translation of rules "*on receiving MessageIn check Condition update State send MessageOut*" into $\mathscr{E}_{hhf}$ is shown. Messages can have any form.

Team working. The attention devoted to teamwork in a MAS setting is quite recent. For this reason, none of the languages discussed so far encapsulates explicit constructs for team specification and programming. The developer can define protocols for forming teams and she/he can try to program agents which respect the given protocol. According to the support given to communication (previous paragraph), the task of defining such protocols may be more or less difficult.

With respect to the six languages we analyzed in this paper, the only papers explicitly addressing the problem of forming groups are by Fisher and Kakoudakis (2000) and Fisher (1998) dealing with flexible grouping in Concurrent METATEM. In Concurrent METATEM, a group is essentially a set consisting of both agents and further sub-groups. The basic properties of groups are that agents are able to broadcast a message to the members of a group, add an agent to a group, ascertain whether a certain agent is a member of a group, remove a specified agent from a group, and construct a new subgroup.

We are not aware of similar extensions of ConGolog, AGENT-0, IMPACT, DyLOG and $\mathscr{E}_{hhf}$.

Heterogeneity and knowledge sharing. Agents programmed in the same language *have the potential* to interact without respecting any specific condition. Whether the agents *will* interact or not depends on the correctness of their code with respect to the specification of the application. Whatever the language used is, if agent A sends a message *Message* to agent B and the code of agent B does not include rules (or imperative statements, or clauses) for managing *Message*, A and B will not be able to engage in a dialog.

Among the six languages discussed in this paper, IMPACT is the most suitable one to cope with heterogeneity of data sources. Any information source or software application can be accessed through an IMPACT program, provided it is properly "agentified". IMPACT can be seen as a programming layer providing a uniform access to heterogeneous sources of information and applications.

The other five languages are not conceived for accessing heterogeneous data sources and for integrating the information contained in the data sources: they provide no support for these two tasks.

In all of the six languages, agents *may* take advantage of sharing the same ontology to interact, but they *are not forced* to do so. To make an example, DyLOG agents can refer to a common ontology contained in the domain knowledge. This approach is described in Baldoni *et al*. (2003). However, it is also possible to develop DyLOG agents without explicitly defining a common ontology. When developing a MAS, the developer has in mind the ontology the agents will refer to. Although it is a good practice to make it explicit, this is not compulsory to guarantee the MAS working.

Programming style. The constructs provided by ConGolog allow to mimic both goal-directed reasoning ("*if the current goal is G then call the procedure to achieve G*") and reactiveness ("*interrupt as soon as the condition C becomes true*").

AGENT-0 programming style is reactive: depending on the message received by the agent and on its current beliefs, a commitment rule can be used.

IMPACT implements a forward reasoning mechanism: the interpreter looks for all the action status atoms which are true with respect to the current state, the agent program and the agent integrity constraints.

For DyLOG a goal directed proof procedure is defined, which allows to compute a query from a given dynamic domain description.

Concurrent METATEM is defined as a language for modeling reactive systems (Fisher, 1993). Thus, reactiveness is the predominant feature of Concurrent METATEM agents.

$\mathscr{E}_{hhf}$ supports a Prolog-like goal-directed reasoning mechanism.

All of the six languages are flexible enough to specify/implement agents adhering to different agent models.

Modularity. All the languages described in this paper support modularity at the agent level, since they allow the definition of each agent program separately from the definition of the other agents.

ConGolog and DyLOG both 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 (Shoham, 1993) macros are used for readability sake. The macro expansion mechanism is not supported by the AGENT-0 implementation.

$\mathscr{E}_{hhf}$ 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 for 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. Different interpreters for languages extending or slightly modifying ConGolog have been proven correct with respect to the intended semantics of the language. See, for example, De Giacomo *et al*. (2000), McIlraith and Son (2002) and Son *et al*. (2001).

There are three different semantics which can be associated with 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 than the rational one, which is more refined than the feasible one. All of them are defined as a set of action status atoms of the form $\mathbf{Do}\,\alpha(\vec{t})$ that are true with respect to the agent program $\mathscr{P}$, the current state $\mathscr{O}$ and the set $\mathscr{IC}$ of underlying integrity constraints. In Eiter and Subrahmanian (1999), algorithms for evaluating the semantics of arbitrary agent programs are proposed and their complexity is evaluated; computing the reasonable status set semantics of a proper subset of IMPACT agent programs, called regular agents, is possible in polynomial time, as demonstrated in Eiter *et al*. (2000).

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. DyLOG is defined by a proof procedure which is sound and complete with respect to the Kripke semantics of modal logic.

The semantics of Concurrent METATEM is the one defined for the first-order temporal logic FML. It is a Kripke-style semantics given by the $\models$ relation that assigns the truth value of a formula in a model $\mathscr{M}$ at a particular moment in time i and with respect to a variable assignment. The soundness, completeness and termination of the resolution procedure discussed in Fisher (1991) have been established.

The $\mathscr{E}_{hhf}$ 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. Soundness and completeness results are established with respect to linear logic.

10 Related work

To the best of our knowledge, there are only few previous attempts to analyze and compare a large set of logic-based formalisms and calculi for multiagent systems.

Some issues such as Kripke models and possible world semantics, Shoham's AGENT-0, Concurrent METATEM etc. are briefly surveyed in Wooldridge and Jennings (1995), but not specifically in a logic-based perspective.

A discussion on the adoption of logic programming and non-monotonic reasoning for evolving knowledge bases (and, more in general, for intelligent agents) can be found in Leite (2003). The book focuses on logic programming for non-monotonic reasoning and on languages for updates. It analyzes and compares LUPS (Alferes *et al*., 2002), EPI (Eiter *et al*., 2001), and introduces their extensions KUL and KABUL. For some of the languages discussed in Leite (2003) a working interpreter exists: see the Implementations of Logic Programs Updates Home Page (2002) for details. We will shortly describe LUPS and KABUL in the sequel as representative examples of languages of updates.

The work which shares more similarities with ours is the paper "*Computational Logic and Multi-Agent Systems: a Roadmap*" (Sadri and Toni, 1999). That paper discusses different formalisms with respect to the representation of the agent's mental state, the agent life-cycle, the ability to communicate following complex interaction protocols and the capability of representing and reasoning about other agents' beliefs. The languages

and systems analyzed in Sadri and Toni's roadmap include INTERRAP (Müller *et al*., 1998; Jung and Fisher, 1997; Müller, 1996), 3APL (Hindriks *et al*., 1998a; Dastani *et al*., 2003), and the work by many others (Kowalski and Sadri, 1999; Shanahan, 2000; Baral and Gelfond, 2000; Pereira and Quaresma, 1998; Gelder *et al*., 1988; Dell'Acqua *et al*., 1998, 1999; Dell'Acqua and Pereira, 1999; Hindriks *et al*., 1999; Carbogim and Robertson, 1999; Poole, 1997). Our paper complements Sadri and Toni's survey because, apart from the IMPACT language and part of the work on CaseLP which are also discussed by Sadri and Toni, we analyze different languages and approaches from different perspectives. Sadri and Toni mainly aim at putting in evidence the contribution of logic to knowledge representation formalisms and to basic mechanisms and languages for agents and MAS modeling. Our paper analyzes a subset of logic-based executable languages whose main features are their suitability for specifying agents and MASs and their possible integration in the ARPEGGIO framework. We think that a researcher interested in logic-based approaches to multiagent systems modeling and prototyping can find an almost complete overview in reading both Sadri and Toni's paper and ours.

Other relevant logic-based approaches that are dealt with neither in Sadri and Toni (1999), nor in this work are ALIAS, LUPS, KABUL, AgentSpeak(L) and the KARO framework. Given more time and space, they could fit in the picture and in the future we would like to check the feasibility of incorporating some of them in the ARPEGGIO framework.

ALIAS (*Abductive LogIc AgentS* (Ciampolini *et al*., 2003)) is an agent architecture based on intelligent and social logic agents where the main form of agent reasoning is abduction. ALIAS agents can coordinate their reasoning with other agents following several coordination schemas. In particular, they can either cooperate or compete in the solution of problems. The ALIAS architecture is characterized by two separate layers: the lower layer involves reasoning while the upper layer involves social behavior.

Agent reasoning is specified by abductive logic programs consisting of a set of clauses *Head :- Body*, where *Head* is an atom and *Body* is a conjunction of literals (atoms and negation of atoms), plus a set of abducible predicates and a set of integrity constraints.

Agent interaction and coordination is specified in a logic language named LAILA (*Language for AbductIve Logic Agents*) suitable for modeling agent structures from the viewpoint of social behavior. LAILA provides high-level declarative operators such as the communication operator $>$, the competition operator ;, the collaboration operator & and a down-reflection operator $\downarrow$ which allows a local abductive resolution to be triggered. The operational semantics of LAILA is discussed in Ciampolini *et al*. (2003).

The distinguishing features of ALIAS consist of its support for (i) *coordinating the reasoning* of agents at a high level of abstraction, (ii) explicitly referring to the *hypothetical reasoning* capabilities of agents from within the language, and (iii) providing the tools to maintain the system (or part of it) *consistent* with respect to the integrity constraints.

A prototypical version of ALIAS has been implemented on top of Jinni (Jinni Home Page, 2003), a logic programming language extended with primitives for concurrent programming.

The main difference between ALIAS and the six languages discussed in our paper lies in the background logic upon which the languages are based: ALIAS is based on first-order

logic while all the six languages discussed in this paper encapsulate features of linear, modal or temporal extensions of first-order logic.

ALIAS shares with Concurrent METATEM and AGENT-0 the support for communication at the language level. However, the communication primitives provided by ALIAS, associated with a semantics of collaboration and competition and with the local abductive reasoning operator (down-reflection), are more expressive than those provided by Concurrent METATEM and AGENT-0 and more suitable for tackling problems that require the coordination of agent reasoning in presence of incomplete knowledge.

The agent reasoning form (abduction) is a shared feature between ALIAS and DyLOG.

The language LUPS (*"the language for dynamic updates"* (Alferes *et al*., 2002)) is based on a notion of update commands that allow the specification of logic programs. Each command in LUPS can be issued in parallel with other LUPS commands and specifies an update action, basically encoding the assertion or retraction of a logic program rule. An extension to LUPS, EPI (*"the language around"* (Eiter *et al*., 2001)), introduces the ability to access external observations and make the execution of programs dependent on both external observations and concurrent execution of other commands. EVOLP (*EVOlving Logic Programs* (Alferes *et al*., 2002)) integrates in a simple way the concepts of both Dynamic Logic Programming and LUPS.

KABUL (*Knowledge And Behavior Update Language* (Leite, 2003)) overcomes some limitations of LUPS. In particular, it allows the specification of updates that depend on a sequence of conditions ("assert a rule *R* if some condition *Cond1* is true after some condition *Cond2* was true"), delayed effects of actions, updates that should be executed only once, updates that depend on the concurrent execution of other commands, updates that depend on the presence or absence of a specific rule in the knowledge base, and inhibition of a command. Moreover, with respect to LUPS, KABUL provides more flexible means to specify updates that will occur in the future and to deal with effects of actions.

AgentSpeak(L) (Rao, 1996) is a programming language based on a restricted first order language with events and actions. The behavior of the agent is dictated by the programs written in AgentSpeak(L). The beliefs, desires and intentions of the agent are not explicitly represented as modal formulae. The current state of the agent, which is a model of itself, its environment and other agents, is viewed as its current belief state; states which the agent wants to bring about based on its external or internal stimuli can be viewed as desires; and the adoption of programs to satisfy such stimuli can be viewed as intentions. An operational semantics of AgentSpeak(L) is provided, as well as the proof theory of the language. AgentSpeak(XL) (Bordini *et al*., 2002) integrates a task scheduler into AgentSpeak(L) to ensure an efficient intention selection. Hindriks *et al*. (1998b) demonstrate that every agent which can be programmed in AgentSpeak(L) can be programmed in the already cited 3APL language. Hindriks *et al*. (1998b) write that, in their opinion, the converse (simulating 3APL by AgentSpeak(L)) is not feasible and thus they conjecture that 3APL has strictly more expressive power than AgentSpeak(L). A simulation of ConGolog by 3APL has also been provided (Hindriks *et al*., 2000) showing that 3APL and ConGolog are closely related languages.

The framework KARO (*Knowledge, Abilities, Results and Opportunities* (van Linder *et al*., 1995)) formalizes motivational attitudes situated at two different levels. At the *assertion* level (the level where operators deal with assertions), *preferences* and *goals* are dealt with. At the *practition* level (the level where operators range over actions) *commitments* are defined. The main informational attitude of the KARO framework is *knowledge*. The fact that agent i knows φ is represented by the formula $\mathbf{K}_i\varphi$ and is interpreted in a Kripke-style possible worlds semantics. At the action level *results*, *abilities* and *opportunities* are considered. The abilities of an agent are formalized via the $\mathbf{A}_i$ operator: $\mathbf{A}_i\alpha$ denotes the fact that agent i has the ability to do α. Dynamic logic is used to formalize the notions of opportunities and results. $do_i(\alpha)$ refers to the performance of action α by the agent i. The formula $\langle do_i(\alpha)\rangle\varphi$ represents the fact that agent i has the opportunity to do α and that doing α leads to φ. The formula $[do_i(\alpha)]\varphi$ states that if the opportunity to do α is indeed present, doing α results in φ. Starting from these basic attitudes, preferences, goals and commitments can be modeled. Different methods for realizing automated reasoning within agent-based systems modeled using the KARO framework are discussed in Hustadt *et al*. (2001a, 2001b).

11 Conclusion

In this paper we have systematically analyzed and compared six logic-based and executable MAS specification languages. Although these languages were chosen on the basis of their potential to be integrated in the ARPEGGIO framework, they are an interesting and representative set of formalisms based on extensions of first order logic. We have discussed the logic-based formalisms upon which the languages are built to allow the reader to understand the theoretical foundations of the languages. We have demonstrated the use of these languages by means of an example, and we have compared them along twelve dimensions. Finally, we have surveyed other approaches adopting computational logic for MAS specification.

Various advantages in using logic-based approaches for modeling and prototyping agents and MAS should emerge from this paper: as pointed out by Wooldridge and Jennings (1995, Section 2), agents are often modeled in terms of mental attitudes such as beliefs, desires, intentions, choices and commitments. The main advantage in using logic and modal logic in particular for modeling intentional systems is that it allows to easily and intuitively represent intentional notions without requiring any special training. The *possible world* semantics usually adopted for modal languages has different advantages: it is well studied and well understood, and the associated mathematics of "correspondence theory" is extremely elegant. Formal languages that support temporal operators are a powerful means for specifying sophisticated *reactive* agents in a succinct fashion. Moreover, since agents are expected to *act*, languages based on formal theories of action such as dynamic logic and the situation calculus are extremely suitable to model agents' ability to perform actions.

According to the observations above, if we compare logic languages and object-oriented formalisms for the specification of agents we note that logic languages are more suitable than object-oriented languages to model agents. According to Odell (2002), autonomy and interaction are the key features which differentiate agents and objects. Autonomy has

two independent aspects: dynamic autonomy and nondeterministic autonomy. Agents are dynamic because they can exercise some degree of activity, rather than passively providing services. With respect to dynamic autonomy, agents are similar to active objects. By means of the running example we have shown that logic-based languages are suitable for expressing the active behavior of agents in a concise and simple way. Agents may also employ some degree of unpredictable (or nondeterministic) behavior. We have shown that all of the six languages analyzed in this paper support some kind of nondeterminism. The "or" connective and the "exists" quantifier introduce a degree of nondeterminism to all the languages based on first order logic. Interaction implies the ability to communicate with the environment and other entities. Object messages (method invocation) can be seen as the most basic form of interaction. A more complex degree of interaction would include those agents that can react to observable events within the environment. And finally in multiagent systems, agents can be engaged in multiple, parallel interactions with other agents. Logic-based languages prove their suitability in modeling agents that react to an event (logical implications of the form *if the event E took place then something becomes true* can be used for this purpose) and to reason about sophisticated conversations.

The last consideration of our paper deals with the implementation of a MAS prototype: we have seen that different languages among the ones we discussed have an interpreter which extends logic programming in some way. Using a logic programming language for MAS prototyping has different advantages:

- *MAS execution*: the evolution of a MAS consists of a nondeterministic succession of events; from an abstract point of view a logic programming language is a nondeterministic language in which computation occurs via a search process.
- *Meta-reasoning capabilities*: agents may need to dynamically modify their behavior so as to adapt it to changes in the environment. Thus, the possibility given by logic programming of viewing programs as data is very important in this setting.
- *Rationality and reactiveness of agents*: the *declarative* and the *operational* interpretation of logic programs are strictly related to the main characteristics of agents, i.e., *rationality* and *reactiveness*. In fact, we can think of a *pure* logic program as the specification of the rational component of an agent and we can use the operational view of logic programs (e.g. left-to-right execution, use of non-logical predicates) to model the reactive behavior of an agent. The adoption of logic programming for combining reactivity and rationality is described in Kowalski and Sadri (1996).

Acknowledgements

We want to thank Marco Bozzano (ITC - IRST, Trento, Italy) and Giorgio Delzanno (University of Genova, Italy) for their useful suggestions. We also thank Matteo Baldoni (University of Torino, Italy), for his clarification on some aspects related with DyLOG and Paolo Torroni (University of Bologna, Italy) for his support in the description of ALIAS and LAILA. Finally we thank the anonymous referees for their comments which helped to improve the paper.

References

ALCHOURRÓN, C. E. AND BULYGIN, E. 1971. *Normative Systems*. Springer-Verlag.

ALFERES, J. J., BROGI, A., LEITE, J. A. AND PEREIRA, L. M. 2002. Evolving logic programs. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, S. Flesca, S. Greco, N. Leone and G. Ianni, Eds. Spriger-Verlag, 50–61. LNCS 2424.

ALFERES, J. J., PEREIRA, L. M., PRZYMUSINSKA, H. AND PRZYMUSINSKI, T. C. 2002. LUPS: A language for updating logic programs. *Artificial Intelligence 138,* 1–2, 87–116.

ALICE HOME PAGE. 2000. `http://www.di.unito.it/~alice/`.

ÅQVIST, L. 1984. Deontic logic. In *Handbook of Philosophical Logic, Vol II*, D. M. Gabbay and F. Guenther, Eds. Reidel, pp. 605–714.

ARISHA, K., EITER, T., KRAUS, S., OZCAN, F., ROSS, R. AND SUBRAHMANIAN, V. S. 1999. IMPACT: A platform for collaborating agents. *IEEE Intelligent Systems 14,* 2, 64–72.

BALDONI, M. 1998. Normal multimodal logics: Automatic deduction and logic programming extension. PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy.

BALDONI, M., BAROGLIO, C., MARTELLI, A. AND PATTI, V. 2003a. Reasoning about conversation protocols in a logic-based agent language. In *Proceedings 8th National Conference of the Italian Association for Artificial Intelligence (AIIA'03)*, A. Cappelli and F. Turini, Eds. Springer-Verlag. LNAI 2829.

BALDONI, M., BAROGLIO, C., MARTELLI, A. AND PATTI, V. 2003b. Reasoning about self and others: communicating agents in a modal action logic. In *Proceedings 8th Italian Conference on Theoretical Computer Science (ICTCS'03)*, R. Gorrieri, C. Blundo and C. Laneve, Eds. Springer-Verlag. LNCS.

BALDONI, M., BAROGLIO, C. AND PATTI, V. 2003. Applying logic inference techniques for gaining flexibility and adaptivity in tutoring systems. In *Proceedings 10th International Conference on Human–Computer Interaction (HICII'03)*, C. Stephanidis, Ed. Lawrence Erlbaum, pp. 517–521.

BALDONI, M., GIORDANO, L., MARTELLI, A. AND PATTI, V. 1997. An abductive proof procedure for reasoning about actions in modal logic programming. In *Proceedings of the 2nd International Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*, J. Dix, L. M. Pereira, and T. C. Przymusinski, Eds. Springer-Verlag, pp. 132–150. LNAI 1216.

BALDONI, M., GIORDANO, L., MARTELLI, A. AND PATTI, V. 1998. A modal programming language for representing complex actions. In *Proceedings DYNAMICS'98: Transactions and Change in Logic Databases. Held in conjunction with the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pp.1–15. Technical Report MPI-9808, University of Passau. `http://daisy.fmi.uni-passau.de/dynamics/workshop98/proceedings.html`.

BALDONI, M., GIORDANO, L., MARTELLI, A. AND PATTI, V. 2000. Modeling agents in a logic action language. In *Proceeding Workshop on Practical Reasoning Agents. Held in conjunction with the International Conference on Formal and Applied Practical Reasoning (FAPR'00)*. London, UK.

BALDONI, M., GIORDANO, L., MARTELLI, A. AND PATTI, V. 2001. Reasoning about complex actions with incomplete knowledge: A modal approach. In *Proceedings 7th Italian Conference on Theoretical Computer Science (ICTCS'01)*, A. Restivo, S. Ronchi Della Rocca, and L. Roversi, Eds. Springer-Verlag, pp. 405–425. LNCS 2202.

BARAL, C. AND GELFOND, M. 2000. Reasoning agents in dynamic domains. In *Logic-based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic, pp. 257–280.

BARRINGER, H., FISHER, M., GABBAY, D., GOUGH, G. AND OWENS, R. 1990. METATEM: A framework for programming in temporal logic. In *Proceedings 1989 REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J. W. de Bakker, W. P. de Roever and G. Rozenberg, Eds. Springer-Verlag, pp. 94–129. LNCS 430.

BLASS, A. 1992. A game semantics for linear logic. *Annals of Pure and Applied Logic 56*, 183–220.

BORDINI, R. H., BAZZAN, A. L. C., DE O. JANNONE, R., BASSO, D. M., VICARI, R. M. AND LESSER, V. R. 2002. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proceedings 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, C. Castelfranchi and W. L. Johnson, Eds. ACM Press, pp. 1294–1302.

BOZZANO, M., DELZANNO, G. AND MARTELLI, M. 1997. A linear logic specification of Chimera. In *Proceedings DYNAMICS'97: (Trans)Actions and Change in Logic Programming and Deductive Databases . Held in conjunction with the International Logic Programming Symposium (ILPS'97).* `http://daisy.fmi.uni-passau.de/dynamics/workshop97/schedule.html`.

BOZZANO, M., DELZANNO, G., MARTELLI, M., MASCARDI, V. AND ZINI, F. 1999a. Logic programming & multi-agent systems: A synergic combination for applications and semantics. In *The Logic Programming Paradigm: a 25-Year Perspective*, K. Apt, V. Marek, M. Truszczynski and D. Warren, Eds. Springer-Verlag, pp. 5–32.

BOZZANO, M., DELZANNO, G., MARTELLI, M., MASCARDI, V., AND ZINI, F. 1999b. Multi-agent systems development as a software engineering enterprise. In *Proceedings 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, G. Gupta, Ed. Springer-Verlag, pp. 46–60. LNCS 1551.

BUGLIESI, M., DELZANNO, G., LIQUORI, L. AND MARTELLI, M. 2000. Object calculi in linear logic. *Journal of Logic and Computation 10,* 1, 75–104.

CARBOGIM, D. AND ROBERTSON, D. 1999. Contract-based negotiation via argumentation (preliminary report). In *Proceedings Workshop on Multi-Agent Systems in Logic Programming (MAS'99). Held in conjunction with the 16th International Conference on Logic Programming (ICLP'99).* `http://www.cs.sfu.ca/conf/MAS99/papers99.html`.

CASTAÑEDA, H.-N. 1975. *Thinking and Doing. The Philosophical Foundations of Institutions*. Reidel, Dordrecht.

CIAMPOLINI, A., LAMMA, E., MELLO, P., TONI, F. AND TORRONI, P. 2003. Co-operation and competition in ALIAS: a logic framework for agents that negotiate. *Annals of Mathematics and Artificial Intelligence 37,* 1–2, 65–91.

CIANCARINI, P. AND WOOLDRIDGE, M. 2000. Agent-oriented software engineering: The state of the art. In *Proceedings 1st International Workshop on Agent-Oriented Software Engineering (AOSE'00)*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, pp. 1–28. LNCS 1957.

COGNITIVE ROBOTICS GROUP HOME PAGE. 2002. `http://www.cs.toronto.edu/cogrobo/`.

DART, P., KAZMIERCKAZ, E., MARTELLI, M., MASCARDI, V., STERLING, L., SUBRAHMANIAN, V. S. AND ZINI, F. 1999. Combining logical agents with rapid prototyping for engineering distributed applications. In *Proceedings 9th International Conference of Software Technology and Engineering (STEP'99)*, S. Tilley and J. Verner, Eds. IEEE Computer Society Press, pp. 40–49.

DASTANI, M., VAN RIEMSDIJK, B., F. DIGNUM AND MEYER, J.-J. 2003. A programming language for cognitive agents: Goal directed 3APL. In *Proceedings 1st Workshop on Programming Multiagent Systems: Languages, Frameworks, Techniques, and Tools (ProMAS03). Held in conjunction with the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03).* `http://www.cs.uu.nl/ProMAS/acceptedpapers.html`.

DAVIES, W. H. AND EDWARDS, P. 1994. Agent-K: An integration of AOP & KQML. In *Proceedings Workshop on Intelligent Information Agents. Held in conjunction with the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, T. Finin and Y. Labrou, Eds.

DE GIACOMO, G., LESPÉRANCE, Y., LEVESQUE, H. AND SARDIÑA, S. 2002. On the semantics of deliberation in IndiGolog – from theory to implementation. In *Proceedings 8th*

International Conference in Principles of Knowledge Representation and Reasoning (KR'02), D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. A. Williams, Eds. Morgan Kaufmann, pp. 603–614.

DE GIACOMO, G., LESPÉRANCE, Y. AND LEVESQUE, H. J. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence 121*, 109–169.

DELL'ACQUA, P. AND PEREIRA, L. M. 1999. Updating agents. In *Proceedings Workshop on Multi-Agent Systems in Logic Programming (MAS'99). Held in conjunction with the 16th International Conference on Logic Programming (ICLP'99)*. `http://www.cs.sfu.ca/conf/MAS99/papers99.html`.

DELL'ACQUA, P., SADRI, F. AND TONI, F. 1998. Combining introspection and communication with rationality and reactivity in agents. In *Proceedings 6th European Workshop on Logics in Artificial Intelligence (JELIA'98)*, J. Dix, L. Fariñas del Cerro, and U. Furbach, Eds. Springer-Verlag, pp. 17–32. LNAI 1489.

DELL'ACQUA, P., SADRI, F. AND TONI, F. 1999. Communicating agents. In *Proceedings Workshop on Multi-Agent Systems in Logic Programming (MAS'99). Held in conjunction with the 16th International Conference on Logic Programming (ICLP'99)*. `http://www.cs.sfu.ca/conf/MAS99/papers99.html`.

DELZANNO, G. 1997. Logic & object-oriented programming in linear logic. PhD thesis, Università di Pisa, Dipartimento di Informatica. Technical Report TD 2/97.

DELZANNO, G. AND MARTELLI, M. 2001. Proofs as computations in linear logic. *Theoretical Computer Science 258,* 1–2, 269–297.

DENNETT, D. C. 1987. *The Intentional Stance*. MIT Press.

DIX, J., MUNOZ-AVILA, H. AND NAU, D. 2003. IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI 4,* 37, 381–407.

EHHF FTP AREA. 1998. `ftp://ftp.disi.unige.it/person/BozzanoM/Terzo/`.

EITER, T., FINK, M., SABBATINI, G. AND TOMPITS, H. 2001. A framework for declarative update specifications in logic programs. In *Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI'01)*, B. Nebel, Ed. Morgan Kauffmann, pp. 649–654.

EITER, T., MASCARDI, V. AND SUBRAHMANIAN, V. S. 2002. Error-Tolerant Agents. In *Computational Logic: Logic Programming and Beyond – Essays in Honour of Robert A. Kowalski, Part I*, A. Kakas and F. Sadri, Eds. Springer-Verlag, pp. 586–625. LNAI 2407.

EITER, T. AND SUBRAHMANIAN, V. S. 1999. Heterogeneous active agents, II: Algorithms and complexity. *Artificial Intelligence 108,* 1–2, 257–307.

EITER, T., SUBRAHMANIAN, V. S. AND PICK, G. 1999. Heterogeneous active agents, I: Semantics. *Artificial Intelligence 108,* 1–2, 179–255.

EITER, T., SUBRAHMANIAN, V. S. AND ROGERS, T. 2000. Heterogeneous active agents, III: Polynomially implementable agents. *Artificial Intelligence 117,* 1, 107–167.

ESHGHI, K. AND KOWALSKI, R. 1989. Abduction compared with negation as failure. In *Proceedings 6th International Conference on Logic Programming (ICLP'89)*, G. Levi and M. Martelli, Eds. MIT Press, pp. 234–254.

FINGER, M., FISHER, M. AND OWENS, R. 1993. METATEM at work: Modelling reactive systems using executable temporal logic. In *Proceedings 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE'93)*, P. Chung, G. L. Lovegrove, and M. Ali, Eds. Gordon and Breach Publishing, 209–218.

FINGER, M., MCBRIEN, P. AND OWENS, R. 1991. Databases and executable temporal logic. In *Proceedings of the Annual ESPRIT Conference 1991*, Comission of the European Communities, Ed. 288–302.

FISHER, M. 1990. Implementing a prototype METATEM interpreter. Technical report, Department of Computer Science, University of Manchester. SPEC Project Report.

FISHER, M. 1991. A resolution method for temporal logic. In *Proceedings 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, J. Mylopoulos and R. Reiter, Eds. Morgan Kaufmann, pp. 99–104.

FISHER, M. 1992. A normal form for first-order temporal formulae. In *Proceedings 11th International Conference on Automated Deduction (CADE'92)*, D. Kapur, Ed. Springer-Verlag, pp. 370–384. LNCS 607.

FISHER, M. 1993. Concurrent METATEM – A language for modeling reactive systems. In *Proceedings 5th International Conference on Parallel Architectures and Language, Europe (PARLE'93)*, A. Bode, M. Reeve, and G. Wolf, Eds. Springer-Verlag, pp. 185–196. LNCS 694.

FISHER, M. 1994. A survey of Concurrent METATEM – the language and its applications. In *Proceedings 1st International Conference on Temporal Logic (ICTL'94)*, D. M. Gabbay and H. J. Ohlbach, Eds. Springer-Verlag, pp. 480–505. LNCS 827.

FISHER, M. 1997. Implementing BDI-like systems by direct execution. In *Proceedings 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*. Morgan Kaufmann, pp. 316–321.

FISHER, M. 1998. Representing abstract agent architectures. In *Proceedings 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, J. P. Müller, M. P. Singh and A. S. Rao, Eds. Springer-Verlag, pp. 227–241. LNAI 1555.

FISHER, M. AND BARRINGER, H. 1991. Concurrent METATEM processes – A language for distributed AI. In *Proceedings of the 1991 European Simulation Multiconference*, E. Mosekilde, Ed. SCS Press.

FISHER, M. AND GHIDINI, C. 1999. Programming resource-bounded deliberative agents. In *Proceedings 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, T. Dean, Ed. Morgan Kaufmann, pp. 200–205.

FISHER, M. AND GHIDINI, C. 2002. The ABC of rational agent programming. In *Proceedings 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, C. Castelfranchi and W. L. Johnson, Eds. ACM Press, pp. 849–856.

FISHER, M. AND KAKOUDAKIS, T. 2000. Flexible agent grouping in executable temporal logic. In *Intensional Programming II (ISPLIP'99)*, M. Gergatsoulis and P. Rondogiannis, Eds. World Scientific.

FISHER, M. AND OWENS, R. 1995. An introduction to executable modal and temporal logics. In *Proceedings Workshop on Executable Modal and Temporal Logics. Held in conjunction with the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, M. Fisher and R. Owens, Eds. Springer-Verlag, pp. 1–20. LNAI 897.

FISHER, M. AND WOOLDRIDGE, M. 1993. Executable temporal logic for distributed AI. In *Proceedings 12th International Workshop of Distributed AI (IWDAI'93)*, K. Sycara, Ed. pp. 131–142.

FISHER, M. AND WOOLDRIDGE, M. 1997. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems 6,* 1, 37–65.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. 2002. FIPA ACL message structure specification. Approved for standard, 2002-12-06. `http://www.fipa.org/specs/fipa00061/`.

GELDER, A. V., ROSS, K. A. AND SCHLIPF, J. S. 1988. Unfounded sets and the well-founded semantics for general logic programs. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*. ACM Press, 221–230.

GIORDANO, L., MARTELLI, A. AND SCHWIND, C. 1998. Dealing with concurrent actions in modal action logic. In *Proceedings 13th European Conference on Artificial Intelligence (ECAI'98)*, H. Prade, Ed. Wiley, pp. 537–541.

GIORDANO, L., MARTELLI, A. AND SCHWIND, C. 2000. Ramification and causality in a modal action logic. *Journal of Logic and Computation 10,* 5, 626–662.

GIRARD, J. Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–102.

HINDRIKS, K., LESPERANCE, Y. AND LEVESQUE, H. 2000. An embedding of ConGolog in 3APL. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'00)*, W. Horn, Ed. IOS Press, pp. 558–562.

HINDRIKS, K. V., DE BOER, F. S., VAN DER HOEK, W. AND MEYER, J.-J. C. 1998a. Control structures of rule-based agent languages. In *Proceedings 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, J. P. Müller, M. P. Singh, and A. S. Rao, Eds. Springer-Verlag, pp. 381–396. LNAI 1555.

HINDRIKS, K. V., DE BOER, F. S., VAN DER HOEK, W. AND MEYER, J.-J. C. 1998b. A formal embedding of AgentSpeak(L) in 3APL. In *Advanced Topics in Artificial Intelligence*, G. Antoniou and J. Slaney, Eds. Springer-Verlag, pp. 155–166. LNAI 1502.

HINDRIKS, K. V., DE BOER, F. S., VAN DER HOEK, W. AND MEYER, J.-J. C. 1999. Semantics of communicating agents based on deduction and abduction. In *Proceedings Workshop on Agent Communication Languages. Held in conjuction with the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, F. Dignum and B. Chaib-draa, Eds. pp. 105–118.

HIRSCH, B., FISHER, M. AND GHIDINI, C. 2002. Organising logic-based agents. In *Proceedings Second NASA/IEEE Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS II)*.

HOHFELD, W. N. 1913. Fundamental legal conceptions as applied to judicial reasoning. *Yale Law Journal 23*, 16–59.

HUSTADT, U., DIXON, C., SCHMIDT, R. A., FISHER, M., MEYER, J.-J. AND VAN DER HOEK, W. 2001a. Reasoning about agents in the KARO framework. In *Proceedings 8th International Symposium on Temporal Representation and Reasoing (TIME'01)*, C. Bettini and A. Montanari, Eds. IEEE Computer Society, pp. 206–213.

HUSTADT, U., DIXON, C., SCHMIDT, R. A., FISHER, M., MEYER, J.-J. AND VAN DER HOEK, W. 2001b. Verification within the KARO agent theory. In *Proceedings 1st International Workshop on Formal Approaches to Agent-Based Systems (FAABS'00)*, J. L. Rash, C. A. Rouff, W. Truszkowski, D. Gordon and M. G. Hinchey, Eds. Springer-Verlag, pp. 33–47. LNAI 1871.

IMPLEMENTATIONS OF LOGIC PROGRAMS UPDATES HOME PAGE. 2002. `http://centria.di.fct.unl.pt/~jja/updates/`.

JENNINGS, N., SYCARA, K. AND WOOLDRIDGE, M. 1998. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems 1*, 7–38.

JINNI HOME PAGE. 2003. `http://www.binnetcorp.com/Jinni/`.

JUAN, T., MARTELLI, M. MASCARDI, V., AND STERLING, L. 2003a. Creating and reusing AOSE features. `http://www.cs.mu.oz.au/~tlj/CreatingAOSEFeatures.pdf`.

JUAN, T., MARTELLI, M., MASCARDI, V. AND STERLING, L. 2003b. Customizing AOSE methodologies by reusing AOSE features. In *Proceedings 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, J. S. Rosenschein, T. Sandholm, M. Wooldridge and M. Yokoo, Eds. ACM Press, pp. 113–120.

JUNG, C. G. AND FISHER, K. 1997. A layered agent calculus with concurrent, continuous processes. In *Proceedings 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL'97)*, M. P. Singh, A. Rao, and M. Wooldridge, Eds. Springer-Verlag, pp. 245–258. LNAI 1365.

KELLETT, A. AND FISHER, M. 1997a. Automata representations for concurrent METATEM. In *Proceedings 4th International Workshop on Temporal Representation and Reasoning (TIME'97)*. IEEE Press, 12–19. `http://www.computer.org/proceedings/time/7937/7937toc.htm`.

KELLETT, A. AND FISHER, M. 1997b. Concurrent METATEM as a coordination language. In *Proceedings 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, D. Garlan and D. Le Métayer, Eds. Springer-Verlag, pp. 418–421. LNCS 1282.

KOWALSKI, R. AND SADRI, F. 1996. Towards a unified agent architecture that combines rationality with reactivity. In *Proceedings International Workshop on Logic in Databases (LID'96)*, D. Pedreschi and C. Zaniolo, Eds. Springer-Verlag, pp. 137–149. LNCS 1154.

KOWALSKI, R. AND SADRI, F. 1999. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence 25,* 3/4, 391–491.

KOZEN, D. AND TIURYN, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science. Volume B*, J. van Leeuwen, Ed. pp. 789–840.

KRIPKE, S. 1963a. Semantical analysis of modal logic I. Normal propositional calculi. *Zeitschrift fur math. Logik und Grundlagen der Mathematik 9*, 67–96.

KRIPKE, S. 1963b. Semantical considerations on modal logic. *Acta Philosophica Fennica 16*, 83–94.

KRIPKE, S. 1965. Semantical analysis of modal logic II. Non-normal modal propositional calculi. In *The theory of models*, Addison, Henkin, and Tarski, Eds. North-Holland, pp. 206–220.

LEGOLOG HOME PAGE. 2000. `http://www.cs.toronto.edu/cogrobo/Legolog/index.html`.

LEITE, J. A. 2003. *Evolving Knowledge Bases – Specification and Semantics*. IOS Press. Frontiers in Artificial Intelligence and Applications 81.

LESPÉRANCE, Y. AND SHAPIRO, S. 1999. On agent-oriented requirements engineering (position paper). In *Proceedings 1st Workshop on Agent-Oriented Information Systems (AOIS'99)*. `http://www.aois.org/99/PositionPapers.html`.

LEVESQUE, H., PIRRI, F. AND REITER, R. 1998. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science 3,* 18.

LEVESQUE, H. J. AND PAGNUCCO, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings Second International Cognitive Robotics Workshop (CogRob2000). Held in conjunction with the 14th European Conference on Artificial Intelligence (ECAI'00)*. `http://www-i5.informatik.rwth-aachen.de/LuFG/cogrob2000/AcceptedPapers.html`.

LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F. AND SCHERL, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming 31*, 59–84.

MARCU, M., LESPÉRANCE, Y., LEVESQUE, H. J., LIN, F., REITER, R. AND SCHERL, R. 1995. Distributed software agents and communication in the situation calculus. In *Proceedings International Workshop on Intelligent Computer Communication (ICC'95)*, pp. 69–78. `ftp://ftp.cs.toronto.edu/pub/cogrob/distribagents.ps.Z`.

MARINI, S., MARTELLI, M., MASCARDI, V. AND ZINI, F. 2000. Specification of heterogeneous agent architectures. In *Proceedings 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'00)*, C. Castelfranchi and Y. Lespérance, Eds. Springer-Verlag, pp. 275–289. LNAI 1986.

MARTELLI, M., MASCARDI, V. AND ZINI, F. 1999a. A logic programming framework for component-based software prototyping. In *Proceedings 2nd International Workshop on Component-based Software Development in Computational Logic (COCL'99)*, A. Brogi and P. Hill, Eds.

MARTELLI, M., MASCARDI, V. AND ZINI, F. 1999b. Specification and simulation of multi-agent systems in CaseLP. In *Proceedings 1999 Joint Conference on Declarative Programming (AGP'99)*, M. C. Meo and M. Vilares-Ferro, Eds. pp. 13–28.

MASCARDI, V. 2002. Logic-based specification environments for multi-agent systems. PhD thesis, Computer Science Department of Genova University, Genova, Italy. DISI-TH-2002-04.

MAYFIELD, J., LABROU, Y. AND FININ, T. 1995. Evaluation of KQML as an agent communication language. In *Proceedings 2nd International Workshop on Agent Theories, Architectures, and Languages (ATAL'95)*, M. Wooldridge, J. P. Müller and M. Tambe, Eds. Springer-Verlag, pp. 347–360. LNAI 1037.

MCCARTHY, J. 1963. Situations, actions and causal laws. Technical report, Stanford University. (Reprinted in *Semantic Information Processing*, M. Minsky Ed., MIT Press, 1968, pp. 110–117.)

MCILRAITH, S. AND SON, T. C. 2002. Adapting Golog for composition of semantic web services. In *Proceedings 8th International Conference in Principles of Knowledge Representation and Reasoning (KR'02)*, D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. A. Williams, Eds. Morgan Kaufmann, pp. 482–496.

MEYER, J.-J. C. AND WIERINGA, R. J. 1993. *Deontic logic in Computer Science*. Wiley.

MILLER, D. 1996. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science 165,* 1, 201–232.

MÜLLER, J. P. 1996. *The Design of Autonomous Agents – A Layered Approach*. Springer-Verlag. LNAI 1177.

MÜLLER, J. P., FISCHER, K. AND PISCHEL, M. 1998. A pragmatic BDI architecture. In *Readings in Agents*, M. N. Huhns and M. P. Singh, Eds. Morgan Kaufman, pp. 217–225.

ODELL, J. 2002. Objects and agents compared. *Journal of Object Technology 1,* 1, 41–53.

ODELL, J., PARUNAK, H. V. D. AND BAUER, B. 2000a. Extending UML for agents. In *Proceedings 2nd Workshop on Agent-Oriented Information Systems (AOIS'00). Held in conjunction with 17th National conference on Artificial Intelligence (AAAI'00)*, G. Wagner, Y. Lespérance, and E. Yu, Eds. pp. 3–17.

ODELL, J., PARUNAK, H. V. D. AND BAUER, B. 2000b. Representing agent interaction protocols in UML. In *Proceedings 1st International Workshop on Agent-Oriented Software Engineering (AOSE'00)*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, pp. 121–140. LNCS 1957.

OWRE, S., RAJAN, S., RUSHBY, J. M., SHANKAR, N. AND SRIVAS, M. K. 1996. PVS: Combining specification, proof checking, and model checking. In *Proceedings Computer-Aided Verification, CAV'96*, R. Alur and T. A. Henzinger, Eds. Springer-Verlag, pp. 411–414. LNCS 1102.

PEREIRA, L. M. AND QUARESMA, P. 1998. Modelling agent interaction in logic programming. In *Proceedings 11th International Conference on Applications of Prolog (INAP'98)*, S. Fukuda, Ed.

PETRIE, C. J. 2000. Agent-based software engineering. In *Proceedings 1st International Workshop on Agent-Oriented Software Engineering (AOSE'00)*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, pp. 59–76. LNCS 1957.

PIRRI, F. AND REITER, R. 1999. Some contributions to the metatheory of the situation calculus. *Journal of the ACM 46*, 325–361.

POOLE, D. 1997. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94,* 1–2, 7–57.

RAO, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, W. V. de Velde and J. W. Perram, Eds. Springer-Verlag, pp. 42–55. LNAI 1038.

RAO, A. S. AND GEORGEFF, M. 1995. BDI agents: from theory to practice. In *Proceedings 1st International Conference on Multi Agent Systems (ICMAS'95)*, V. Lesser, Ed. AAAI Press, pp. 312–319.

REITER, R. 2001. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic (TOCL) 2,* 4, 433–457.

SADRI, F. AND TONI, F. 1999. Computational logic and multi-agent systems: a roadmap. Technical report, Department of Computing, Imperial College, London.

SHANAHAN, M. P. 2000. Reinventing Shakey. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic, pp. 233–253.

SHAPIRO, S., LESPÉRANCE, Y. AND LEVESQUE, H. J. 1998. Specifying communicative multi-agent systems. In *Agents and Multi-Agent Systems – Formalisms, Methodologies, and Applications*, W. Wobcke, M. Pagnucco and C. Zhang, Eds. Springer-Verlag, pp. 1–14. LNAI 1441.

SHAPIRO, S., LESPÉRANCE, Y. AND LEVESQUE, H. J. 2002. The cognitive agent specification language and verification environment for multiagent systems. In *Proceedings 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, C. Castelfranchi and W. L. Johnson, Eds. ACM Press, pp. 19–26.

SHOHAM, Y. 1993. Agent-oriented programming. *Artificial Intelligence 60*, 51–92.

SON, T. C., BARAL, C. AND MCILRAITH, S. 2001. Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs. In *Proceedings AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, A. Provetti and T. C. Son, Eds. AAAI Press, pp. 202–209.

SUBRAHMANIAN, V., BONATTI, P., DIX, J., EITER, T., KRAUS, S., ÖZCAN, F. AND ROSS, R. 2000. *Heterogenous Active Agents*. MIT Press.

THE TEAMCORE RESEARCH GROUP HOME PAGE. 2003. `http://teamcore.usc.edu/tambe/agent.html`.

THOMAS, S. R. 1995. The PLACA agent programming language. In *Proceedings 1st International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, M. Wooldridge and N. R. Jennings, Eds. Springer-Verlag, pp. 355–370. LNCS 890.

VAN LINDER, B., VAN DER HOEK, W. AND MEYER, J.-J. C. 1995. Formalising motivational attitudes of agents: On preferences, goals and commitments. In *Proceedings 2nd International Workshop on Agent Theories, Architectures, and Languages (ATAL'95)*, M. Wooldridge, J. P. Müller and M. Tambe, Eds. Springer-Verlag, pp. 17–32. LNAI 1037.

VON WRIGHT, G. H. 1951. Deontic logic. *Mind 60*, 1–15. (Reprinted in G. H. von Wright, *Logical Studies*, pp. 58–74. Routledge and Kegan Paul, 1957.)

WEISS, G. 1999. *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

WOOLDRIDGE, M. 2000. *Reasoning about Rational Agents*. MIT Press.

WOOLDRIDGE, M. AND JENNINGS, N. R. 1995. Intelligent agents: Theory and practice. *The Knowledge Engineering Review 10,* 2, 115–152.

ZINI, F. 2000. CaseLP, a rapid prototyping environment for agent-based software. PhD thesis, Computer Science Department of Genova University, Genova, Italy. DISI-TH-2001-03.