From pure Prolog to logic Agent-Oriented Programming Languages

Rafael Bordini[3], Stefania Costantini[1,4], Andrea Monaldini[1,2] and Alina Vozna[1,2]

[1]Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy
[2]University of Pisa, Largo B. Pontecorvo, Pisa, Italy
[3]School of Technology, PUCRS, Porto Alegre – RS, Brazil
[4]Gruppo Nazionale per il Calcolo Scientifico - INdAM, Rome, Italy

Abstract

We motivate and compare Agent-Oriented Logic Programming languages (AOLPs), showing that they are not really a departure from the basic logic programming paradigm, but rather constitute an extension to account for abstractions that are essential to model autonomous agents and multi-agent systems. So, existing AOLPs like AgentSpeak and DALI, which have already been successfully applied in many kinds of applications, can contribute to the spread of logic programming thinking in the next years.

Keywords

Agent-Oriented Programming Languages, Logic Languages, Comparison Among Languages

1. Introduction

Agent-Oriented Programming (AOP) is a programming paradigm that conceptualizes software systems as composed of autonomous, interactive entities known as agents. These agents are designed to operate independently, possess their own goals, and can make decisions to achieve these goals without direct external control. This autonomy allows agents to proactively pursue their objectives, as opposed to merely reacting to external stimuli, which distinguishes AOP from other paradigms like Object-Oriented Programming (OOP) where objects typically react to method calls. Additionally, agents in AOP are characterized by their social ability; they can communicate, negotiate, and collaborate with other agents to form a coherent, functioning system. This makes AOP particularly suitable for developing complex, distributed systems where flexibility, modularity, and scalability are paramount. For instance, AOP has found significant applications in areas such as multi-agent systems (MAS), artificial intelligence, and simulation of real-world phenomena, where systems can benefit from decentralised control and adaptive behaviour [1]. However, the paradigm also presents challenges, including the complexity of designing effective agent interactions and ensuring robust coordination among agents. Despite these challenges, the adoption of AOP continues to grow, driven by the increasing need for sophisticated, autonomous systems in diverse fields.

A wide variety of agent-oriented programming languages and frameworks have emerged since 2005. [2] presents a systematic review of 395 articles spanning a period of time ranging from 2005 to the present. The aim of the authors is to highlight the use trends of programming languages in the field of agent programming, as well as their applications. The graph shows that Java is the most widely used language and that Python has become the most popular language in recent years.

Java became the industry standard language for agent-oriented programming between 2006 and 2016. Because of its adaptability, robustness, and general acceptance, Java has become the preferred language for creating intelligent software agents in a variety of fields and applications. With the help of its vast

WOA 2024: 25th Workshop "From Objects to Agents", July 8-10, 2024, Forte di Bard (AO), Italy
rafael.bordini@pucrs.br (R. Bordini); stefania.costantini@univaq.it (S. Costantini); andrea.monaldini@student.univaq.it (A. Monaldini); alina.vozna@student.univaq.it (A. Vozna)
0000-0001-8688-9901 (R. Bordini); 0000-0002-0877-7063 (S. Costantini); 0009-0004-2518-2055 (A. Monaldini); 0009-0009-0179-6948 (A. Vozna)

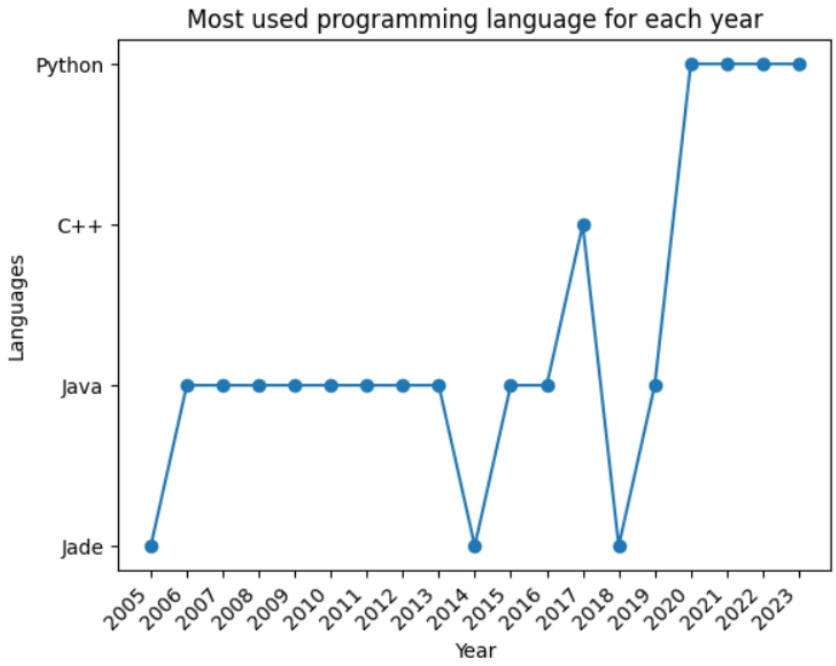

Figure 1: Most used programming language for each year

ecosystem of libraries, frameworks, and tools, developers were able to construct sophisticated agent systems that could address a variety of challenges and specifications [3].

Python is a popular, versatile programming language that is well-known for being easy to learn, flexible, and simple. Since its 1991 debut, Python has become a mainstay in several fields, including scientific computing, data science, web development, and artificial intelligence. Beginners can easily grasp it thanks to its simple and straightforward syntax, and experienced developers can access its advanced features and libraries [4].

This paper will, however, devote particular attention to computational logic due to its potential for verifiability and explainability so that, overall, it can be a suitable tool for trustworthy Artificial Intelligence and Neuro-symbolic applications. In particular, we consider two well-known implemented agent-oriented logic programming languages that have been widely used, also in industrial applications, in recent decades. We compare them with each other and with other recently-published approaches authored by "founding fathers" of logic programming.

The remainder of this paper is structured as follows. The following section compares and contrasts reactive and proactive models with deductive reasoning agents, while the development of logical agent-oriented languages is examined. Section 3 describes AgentSpeak(L), a logic-based language that supports the modeling of agents in dynamic environments that receive perceptual data and act according to their internal mental states. Section 4 introduces DALI, a Prolog-based agent-oriented language designed for logical agents and multi-agent systems. In Section 5, an actual example of a cleaning robot is used to compare DALI to AgentSpeak. It draws attention to some of their commonalities and peculiarities. Section 6 introduces the Evolutionary Semantics approach to provide a declarative semantics for agent-oriented languages like DALI and AgentSpeak. The related work section provides an overview of other various agent-oriented logic languages and a comparison with the recent proposals of LPS and Epilog. Finally, in Section 8 we conclude.

2. Logic Agent-Oriented Languages

The original perspective on agents in Artificial Intelligence was focused on the agents' reasoning process ("deductive reasoning agents"), identifying "intelligence" as rationality, thus neglecting the interactions

of the agents with the environment and with other agents. This perspective has been heavily criticized for instance in [5] [6], that adopts in an extreme way the opposite point of view, arguing that "intelligent" behavior results solely from the ability of an agent to react appropriately to changes in its environment ("purely reactive agents").

More generally however, agents can be seen as independent entities interacting both reactively and proactively with a partially observable external environment, where proactivity is the ability to do things on the agent's ows's initiative, in consequence to past interactions with the environment and/or of internal reasoning. The idea of an agent reasoning about what it chooses to do, and via which means, has been the basis of the seminal approach of the BDI (Belief, Desires, Intention) logic for modelling agents by [7], that resulted in the definition of the AgentSpeak agent-oriented logic programming language [8]. At the same time, in the approach of [9], agents were theories (logic programs), each one with its name, and they were able to communicate with each other via two communication primitives (tell/told). A view of logical agents, able to be both rational and reactive, i.e., capable not only of reasoning and communicating, but also of providing timely response to external events was introduced in [10, 11, 12]. Many significant attempts have been made over time to integrate rationality with reactivity and proactivity in logic programming, see for instance [13], [14], [15], [16] and [17] for a discussion. Along this line came the proposal of "Active Logic Programming"introduced in [18].

After those seminal approaches, both the notion of agency and its interpretation in computational logic have greatly evolved. Several computational-logic-based agent-oriented languages and frameworks to specify agents and Multi-Agent Systems (MAS), that we may call Agent-Oriented Logic Programming languages (AOLPs), have in fact been defined over time (for a survey of these languages and architectures the reader may refer, among many, to [19, 20, 21]). Their added value with respect to non-logical approaches is to provide clean semantics, readability and verifiability, as well as transparency and explainability 'by design' (or almost), as logical rules can easily be transposed into natural-language explanations.

Prolog has proved over time to be a good knowledge representation language, also for modelling intelligence agents. In fact, Prolog has been the basis for the specification of many of the above-mentioned approaches. Two Prolog-based AOLPs are DALI [22, 23] and the Jason variant of AgentSpeak [24]. These two languages have proved to be remarkably successful in practical (even industrial), and cognitive robotics applications. Their distinctive features involve several kinds of events as first-class objects, along with the possibility of explicitly modeling reactivity ad proactivity via special rules. In the following, we argue that these languages are a natural evolution of Prolog, as in fact they can be endowed with a fully logical semantics, the Evolutionary Semantics; in this semantic approach, summarized below, reactive and proactive rules are reinterpreted in terms of standard Horn Clause rules, and results of an agent's interaction with the environment and of the agent's internal proactive operations are interpreted as a sequence of program transformation steps, so as to reason about the "state" of an agent, without introducing explicitly such a notion. Success that AgentSpeak and DALI have achieved in practice, has contributed and in our opinion has the potential to contribute even more in the future to the spread of the logic programming paradigm.

3. AgentSpeak

The Belief-Desire-Intention (BDI) architecture is one of the best known models for the development of intelligent agents and in particular practical reasoning agents. In the BDI approach [7, 8], agents are systems that are situated in a changing environment, receive perceptual input, and take actions to change their environment, based on their internal "mental state". Implementations of BDI agents are being used successfully in real application domains. One of the first concrete platforms based on BDI architecture was PRS [25], applied in several significant multi-agent applications. Several agent-oriented programming languages have been developed based on BDI (surveys can be found here [26] [27]). Among them, one of the best known languages is AgentSpeak(L) [28] — an abstract language based on an concepts of the PRS architecture but heavily influenced by Prolog: AgentSpeak rules, called 'plans' are effectively

guarded Horn clauses. That abstract language was later extended and implemented by concrete Agent Programming platforms such as Jason [29] [30] and ASTRA [31].

AgentSpeak(L) is a logic language with external events and actions, and language construct that are meant to (indirectly) model BDI features in a simple way. The internal state of an AgentSpeak agent is constituted by its beliefs, the goals (i.e., desires), and instances of plans (from a dynamic plan library) for achieving goals are its intentions. External events are interpreted as belief changes, which are pursued concurrently with goals in some order (according to a selection function) by means of plans (selected by another special function). A plan can set new goals, called internal events. Below is an example, concerning an agent controlling a cleaning robot.

$$+location(waste, X) \ : \ location(robot, X) \ \& \ location(bin, X) \\ <- \ pick(waste); drop(waste).$$

$$+location(waste, X) \ : \ location(robot, X) \ \& location(bin, Y) \ \& \ not \ X = Y \\ <- \ pick(waste); \ !location(robot, Y); \ drop(waste).$$

In both rules (in AgentSpeak called "plans"), $+location(waste, X)$ is an *external event*, i.e., a perception of the agent about some aspect of the state of the agent's external environment, meaning that there is waste at some location X (where X is instantiated to some value, as external events are supposed to be represented by ground atoms). After the semicolon, there is a conjunction of Prolog-like subgoals (with $\&$ as a separator) that, if successful, enables the body of the rule, i.e., the part after the $<-$ (that takes the place of Prolog's $:-$) to be executed. The body consists of a sequence of atoms (separated by ;) where each plain one represents an action and each one prefixed by ! represents a new goal, called an *internal event* that the agent itself sets, to be coped with by another rule. The internal event occurring in the above program fragment is coped with by the following two rules/plans:

$$+!location(robot, X) \ : \ location(robot, X) \ <- \ true. \\ +!location(robot, X) \ : \ location(robot, Y) \ \& \ not \ X = Y \\ <- \ !move(robot, Y, X).$$

According to the first rule, if the agent wants to reach location X ($+!location(robot, X)$) and the robot believes to be indeed at location X, nothing needs to be done. According to the second one, if instead the robot believes to be at location Y different from X (where not is Prolog's negation as failure), then the robot will set the goal $!move(robot, Y, X)$ to go to location X. This internal event is managed by rules not shown here.

4. DALI

DALI [22, 23, 32, 33, 34, 35] is a superset of Prolog in the sense that a Prolog program is a DALI program as well. There are however some additional specific features to make the language agent-oriented. In fact, DALI has been specifically devised to program logical agents and multi-agent systems. In DALI, the autonomous behavior of an agent is triggered by several kinds of events: external, internal, present and past events. Reaction to external and internal events is defined by *reactive rules*, indicated by special token $:>$ instead of $:-$.

External events correspond to the perceptions concerning the agent's environment, that arrive to the agent via some kind of sensors. They are indicated with postfix E. Below is an example of an external event, with the corresponding reactive rule.

$$alarm\_clock\_ringsE :> stand\_upA.$$

Precisely, the sound of the alarm clock captured by a sensor is transformed, via some interface we do not care about, into the atom $alarm\_clock\_ringsE$, which is added to the agent's input queue. Atom $stand\_upA$ indicates an action (within the repertoire of those that the agent is capable to do by means of some actuator), in this case without preconditions. Let us say that we are modeling a personal assistant agent, which will advise the user what to do via a natural-language interface, thus $stand\_upA$ will be translated via a suitable interface into a natural-language message. There is only one reactive rule for

every external event, with no loss of generality because the body of a reactive rule may contain any kind of Prolog subgoal, as seen below.

$alarm\_clock\_ringsE{:>}\ switch\_off\_alarm\_clockA,$
$\qquad\qquad\qquad\qquad weekday(Today), choose(Today).$
$choose(Today) :- working\_day(Today), stand\_upA.$
$choose(Today) :- vacation(Today), go\_back\_to\_sleepA.$

Let us now consider a "butler" agent, able to manage the door.

$visitor\_arrived :- bell\_ringsN.$
$bell\_ringsE{:>}\ open\_doorA.$

In this case, $bell\_ringsN$ indicates a *present event* (N for 'now'), i.e., an event that is in the incoming queue but has not been reacted to. As seen, the present event is used to draw an internal conclusion. Upon reaction, specified in the second rule, the event is removed from the queue. The conclusion $visitor\_arrived$ can be interpreted as an *internal event*, by adding the reactive rule specifying that the owners of the house should be told about the visitor:

$visitor\_arrivedI{:>}\ warn\_landlords.$

But, how is the conclusion $visitor\_arrived$ obtained? The interpreter categorizes $visitor\_arrived$ as an internal event due to the presence of the reactive rule. Thus, an internal event is characterized by a couple of rules: the first one is a plain rule, and the second one is a reactive rule. For each internal event like, e.g., $visitor\_arrived$, the interpreter checks the event, that is, queries itself with $?{-}\ visitor\_arrived$ at a certain frequency, and, upon success, will add the corresponding event $visitor\_arrivedI$ (I standing for 'internal') in an incoming queue from which it will be taken to trigger the reactive rule. According to Kowalski's maxim "Algorithm = Logic + Control", the frequency for checking an internal event can be customized by changing the default one, by means of a user directive associated to the program; it is also possible to state conditions to start and/or to stop the check (the default is "check forever"). Internal events make DALI agents proactive, i.e., able to perform inferences and take initiatives without direct external intervention, initiated and performed on the agent's own accord in consequence to internal reasoning.

The DALI interpreter in its present form performs an interleaving of different activities: normal Prolog-like processing; check an internal event and, if successful, insert the event into a dedicated queue; extract an external or internal event from the dedicated queue, and execute the reactive rule. A more powerful interpreter might perform the three activities in parallel. It might even be feasible to parallelize reaction to events. Events are in fact time-stamped, and the order might be of importance. However, while at present the only user directive for events concerns their priority (high priority implies faster reaction), one might allow the programmer to (optionally) specify a partial ordering among events: those unrelated w.r.t. this ordering might be managed in parallel.

Notice that an action may have preconditions, specified by plain Prolog rules that however, for the sake of clarity, are indicated with the special token $:<$. For instance (where not is Prolog's negation-as-failure):

$open\_doorA{:<}\ have\_key.$
$open\_doorA :< not\ have\_key, get\_key.$

A DALI agent remembers to have reacted to an external or internal event, and to have performed an action, by converting the event/action into a *past event*, indicated with postfix P. Past events are exploited in the example below.

$bell\_ringsE{:>}\ open\_doorA.$
$open\_doorA{:<}\ door\_closed.$
$door\_closed :- close\_doorP.$
$close\_doorA{:<}\ door\_open.$
$door\_open :- open\_doorP.$
$unsafe\_situation :- alarm\_reportedP.$
$unsafe\_situationI{:>}\ close\_doorA.$

Here, the butler agent is responsible to open and close a door on which we suppose it has the full control. The agent will, upon need, open and close the door. Action $open\_doorA$ will be actually performed only if the agent remembers to have closed the door before, and thus the door is indeed closed; this is the case if the past event $close\_doorP$ is present in the agent's memory (in practice, it is recorded as a fact added to the agent program). Symmetrically for action $close\_door$, performed only if $open\_doorP$ is in the memory. In addition, the agent will close the door in case there has been an alarm (recorded by other rules as a past event). Notice however that, upon completion of the action $open\_doorA$ (resp. $close\_doorA$) the fact $close\_doorP$ (resp. $open\_doorP$) must be removed, as it represent a situation which is no longer actual. This is done via the user directives:

$keep\ close\_doorP\ until\ open\_doorA.$
$keep\ open\_doorP\ until\ close\_doorA.$

Other user directives manage the agent's memory (which consists in the set of past events), specifying for instance conditions for a past event to expire. Notice that, in case of arrival of similar events (e.g., different measurements of the temperature) or of repeatedly performing the same actions, different past events are created. In fact, all events in DALI are time-stamped, in the sense that they are of the form $t : p(a_1, \ldots, a_n)$ where t is an integer number representing a time instant, and $p(a_1, \ldots, a_n)$ is a ground atoms (i.e., an atom not containing variables); t can be seen as an additional argument of p, but in the proposed syntactic form the time-stamp can be omitted if not needed. Consequently, analogous past events will be all recorded, with a different time-stamp. The interpreter, for each kind of event (i.e., the temperature), keeps the indication of the 'actual' one, meaning the most recent version (the one with the newest time-stamp). So, when a past event is 'removed' by a directive, it is simply removed from the set of the actual ones. Notice that the set of actual past events constitutes the best possible agent's approximation of the present situation concerning the external environment and the agent itself. The 'old' past events, that can be accessed via their time-stamps, might be useful for reasoning, think, e.g., of the calculation of the average/maximum/minimum temperature in a certain period. Time-stamped past events allow one to define fluents over intervals, e.g., the following example defines a fluent and computes the actual time intervals where the fluent holds, considering implicitly the last version of past events (where $notevP$ means that no version of past event evP is present in the memory):

$door\_closed(T, T1) :- close\_doorP : T,\ open\_doorP : T2, T2 >= T1.$
$door\_closed(T, T1) :- close\_doorP : T,\ now(Tn), T1 =< Tn,\ not\ open\_doorP.$

The usefulness of time-stamps is also seen in the example below, modeling an agent that checks for attacks of some kind (e.g., cyberattacks):

$$alarmE(K) : T :> alarmP(K) : T1,\ T1 - T < threshold1,\ alert\_operatorA.$$
$$alarmE(K1) : T1,\ alarmE(K2) : T2 :>\ T1 - T2 < threshold2,\ K1 = / = K2,\ close\_accessA,\ alert\_operatorA.$$

The first reactive rule detects an alarm of the same kind of a past alarm occurred recently; notice the difference between the external event $alarmE(K) : T$ happening now and the past event $alarmP(K) : T1$ corresponding to an external event happened previously, both of them time-stamped. The second rule detects two external events of the same kind occurring in conjunction. In fact, it can be seen that, in the head of the second reactive rule, there are two external events supposed to occur together. A user directive will state what 'together' means, specifying a time interval in which both timestamps should be included.

DALI provides features for basic planning tasks. In particular, postfix G indicates a so-called DALI 'goal', i.e., an internal event that, once successful and reacted to, expires, in the sense that it will no longer be attempted. However, to manage more involved planning activities, an Answer Set Solver (cf. [36] and references therein for Answer Set Programming) has been integrated into the implementation, and can be invoked by a DALI agent. The use of the 'goal' construct is seen in the example below, when the butler agent will prepare a cake (only once) if some children are expected to visit. The example includes a goal and a 'normal' internal event. The variable $Cake$ will have as a value the specification

of the kind of cake one wants to prepare. The example overcomes the baking procedure. The cake is deemed to be good and ready to eat upon heuristic conditions.

$prepare\_cakeG(Cake) :- children\_visitors\_expected,\ favorite\_children\_cake(Cake).$
$prepare\_cakeG(Cake) :> bakeA(Cake),\ put\_in\_ovenA(Cake),\ ready(Cake).$
$check\_ready(Cake) :- put\_in\_ovenP(Cake),$
$\quad color(Cake, golden), smell(Cake, good).$
$check\_readyI(cake) :> take\_from\_ovenA(cake),\ switch\_off\_ovenA, eatA(cake).$

5. Comparison between DALI and AgentSpeak

Below we show a DALI program for the cleaning robot, analogous (with some additions) to the AgentSpeak program illustrated before, and we discuss the analogies and differences.

To begin with, let's notice that to ascertain its most recent location, the DALI robot exploits an internal event. The first rule is attempted at a certain frequency to detect the actual position by the robot's sensors (frequency will affect, here, the precision with which the robot knows its position); the second (reactive) rule will do nothing, having however the effect to record the most recent position as a past event. Low level of battery charge, as an external event returned by a internal sensor, is managed, via a related reactive rule, by going to the recharge station; the min threshold for the battery will be devised by the programmer so as to be sufficient to reach the station.

$location(robot, L) :- check\_sensors(L).$
$locationI(robot, L) :> true.$
$battery\_levelE(robot, B) :>\ B =< min,\ locationP(robot, L),$
$\quad location(charger, D),\ moveG(robot, L, D),$
$\quad rechargeA.$

$moveG(robot, L1, L2) :- L1 = \backslash = L2.$
$moveG(robot, L1, L2) :> \ldots$

In the second reactive rule, $locationP(robot, L)$ means that the robot remembers to be at location L (last location detected), and $location(charger, D)$ means that the agent believes that the location of the charger is D; presumably, this is a fact present since the beginning in the agent program. Notice that $moveG(robot, L, D)$ is a DALI so-called "goal", i.e., a special internal event which is executed only once as soon as invoked (no associated frequency here); this one, in particular, is aimed to get the robot moving from a location to another one. The first rule, which enables the internal event, checks that the starting and destination locations are different – otherwise nothing must be done – and if so, proceeds somehow to move in the environment. The code below is analogous to the AgentSpeak one, coping with the fact that DALI admits only one reactive rule for each event. This is not an essential language feature; rather, this choice has been done to make the interpreter simpler. The context as specified in AgentSpeak rules in DALI is represented by the first subgoals in rules bodies.

$locationE(waste, X) :> evaluate(robot, X).$
$evaluate(robot, X) :- locationP(robot, L), L == X, pickupA(waste),$
$\quad location(bin, B)\ , moveG(robot, L, B),\ dropA(waste, bin).$
$evaluate(robot, X) :- locationP(robot, L),\ L = / = X,\ moveG(robot, L, X),$
$\quad pickupA(waste),\ location(bin, B),$
$\quad moveG(robot, X, B),\ dropA(waste, bin).$

It is easy to notice the analogies between the two languages, looking beyond the different syntax. There are some differences, e.g., in Agent Speak we do not have different versions of time-stamped past events, and we do not have DALI-like internal events, i.e., actived proactively at a default or user-defined frequency.

In order to be able to program multi-agent systems, DALI and AgentSpeak have both been made compliant to the FIPA standard, where FIPA is a widely used standardized ACL (Agent Communication Language), cf. http://www.fipa.org/specs/fipa00037/SC00037J.html.

DALI, however, also features a full communication architecture composed of three layers. The first layer implements a FIPA-compliant communication protocol. The second layer allows an agent to filter incoming and outcoming messages via special (optional) meta-rules concerning the distinguished predicates *tell* and *told*. And, the last layer (optionally) employs a meta-interpreter to allow for interoperability by means of exploiting internal or external ontologies. Below is an example concerning message filtering. There, an agent receives a communication informing it about a film F which is now being screened in theaters. The FIPA primitive *inform*, with parameters message content and sender, is received by the agent as the external event $informE(film(F), A)$. However, before being inserted into the external event queue, since there is a *told* rule whose head matches with the message, it is verified that the rule's body is true. If this is not the case, i.e., here, if the sender agent is either not a friend or not trusted (code for deciding not shown here), then the message is simply deleted, so the related reactive rule will never be fired. If the message is accepted, in this program the agent will go to see the film, and will also request to ann to go as well. However, before being inserted into the outcoming messages queue, since there is a *tell* rule whose head matches with the message, it is verified that the rule's body is true. If this is not the case, i.e., here, ann is either not a friend or she is not nice (code for deciding not shown here), then the message is simply deleted and will never be sent. Notice that FIPA message syntax includes some more parameters, that are automatically coped with by the interpreter.

$told(inform(film(F), A)) :- friend(A),\ trusted(A).$
$tell(proposeA(go\_to\_see(F), A)) :- friend(A), nice(A).$
$informE(film(F), A) :> go\_to\_seeA(F),\ proposeA(go\_to\_see(F), ann).$

Notice also that, the same 'core' agent program, if equipped with a different set of *tell/told* rules, will result in an overall agent behaving differently (e.g., bold rather than cautious, friendly rather than stern, etc.).

6. Declarative Semantics of Evolving Agents

In order to demonstrate that DALI and AgentSpeak are indeed en enhancement of Prolog, we provide below some hints about a declarative semantic approach (first presented in [37]) called *Evolutionary Semantics*, which is immediately applicable to both of them. In fact, DALI was designed from the beginning in order to have a declarative semantics. AgentSpeak was not, but the general semantic approach presented below turns out to be applicable to it as well.

The difficulty in defining a semantics for agent-oriented languages is that agents evolve according to the interaction with their external environment, while, traditionally, logic includes neither the notion of state nor that of evolution. The Evolutionary Semantics consider DALI and AgentSpeak programs as Prolog programs where a limited form of assert/retract is allowed. As described in detail in [37], these programs can be in fact translated into plain Prolog programs plus some asserts/retracts. Below the sample translation of a reactive rule is shown (disregarding, for the simplicity, timestamps). Given the reactive rule:

$alarm\_clock\_ringsE :> stand\_upA.$

the translation into Prolog is:

$react\_alarm\_clock\_rings :-$
$\quad alarm\_clock\_ringsE,\ stand\_upA,$
$\quad retract(alarm\_clock\_ringsE),$
$\quad assert(alarm\_clock\_ringsP),\ assert(stand\_upP).$
$stand\_upA :- true.$

The Prolog rule takes profit of the fact that the external event $alarm\_clock\_ringsE$, once arrived, is asserted as a new fact. The action $stand\_upA$, considered as a Prolog subgoal, and having no preconditions, always succeeds. Considering that a past event is always new, being characterized by a timestamp and thus distinguishable from analogous previous events, no destructive update is needed.

At any time, the Prolog counterpart P_{Prol} of agent program P admits the Least Herbrand Model [38] (or the Perfect Model if negation as failure is employed, where every program is assumed to be stratified

[39]). This model changes when a new event is asserted/retracted. Each assert/retract can be seen as a program transformation step. Then, one will have an initial program $P_0 = P_{Prol}$ which, according to these program-transformation steps (each one transforming P_i into P_{i+1}), gives rise to a Program Evolution Sequence $PE = [P_0, ..., P_n]$. The program evolution sequence will have a corresponding Semantic Evolution Sequence $ME = [M_0, ..., M_n]$ where M_i is the semantic account of P_i.

The Evolutionary Semantics ε^{Ag} of agent Ag is thus the tuple $\langle PE, ME \rangle$, over a potentially infinite evolution. The *snaphot at stage i* is the tuple $\langle P_i, M_i \rangle$. The evolutionary semantics represents the evolution of an agent without introducing a formal concept of "state" in the sense of some memory items on which destructive updates are applied. The snapshot denotes the activity of an agent performed up to a certain stage, but none of the previous stages is overwritten, they are encompassed by the Evolutionary semantics. Note that the Evolutionary Semantics allows forms of static or dynamic checking of an agent's behaviour to be performed (cf., e.g., [40, 41, 42, 43]).

Speech Act Theory (SAT) [44, 45, 46] is a theory that aims to understand how utterances can be used to achieve actions, consisting of locutionary acts (uttering a sentence), illocutionary acts (expressing the speaker's intention), and perlocutionary acts (what is achieved by saying something). In agent communication, speech-act theory is adapted to determine how agents interpret messages, with the actual behavior depending on the agent's plan library and circumstances at the time of message processing. Concerning communication, according to the Speech-Act-Theory both DALI and AgentSpeak interpret incoming messages as special events and outcoming messages as special actions, so no semantic extension is needed.

7. Related Work

Both DALI and AgentSpeak were designed in order to understand whether modeling agents in logic programming was possible, and to which extent. These new languages were intentionally kept as simple and easy to understand as the Horn clause language: so, both syntax and semantics are very close to the Horn clause language, and so is the procedural semantics. Therefore, our claim is that they truly constitute agent-oriented versions of Prolog. Thus, they can spread the popularity of logic programming among designers of practical applications of agents.

Notice that DALI and AgentSpeak are meant to be logic general-purpose programming languages like Prolog, aimed at programming agents.They do not commit to any specific agent architecture, and also, they do not commit to any specific planning formalism. Then, they do not directly compare with approaches like ConGolog [47], which is a multi-agent Prolog-like language with imperative features based on situation calculus, and 3APL [48, 49], which is rule-based, planning-oriented, and has no concept of event. Also, a comparison with very extensive approaches for Multi-Agent-Systems like IMPACT [50] is not in order, since IMPACT is not just a language, but proposes a complex agent architecture.

A purely logic language for agents is METATEM [51] [52], where different agents are logic programs which are executed asinchronously, and communicate via message-passing. METATEM has a concept of time, and what happened in the past determines what the agent will do in the future. Differently from DALI and AgentSpeak, METATEM agents are purely reactive, and there are no different classes of events.

We now proceed to discuss 'competing' approaches presented in papers appearing in the 2023 book "Prolog: The Next 50 Years", namely, LPS [53] and Epilog [54]. We propose comparisons on relevant examples showing either DALI or the AgentSpeak counterparts; as we have seen before in fact, the two are almost fully interchangeable.

The paper by Kowalski et al. presents the language LPS (Logic Production Systems) [55] [56] [57], which is procedurally executed via the imperative paradigm of solving goals by generating a sequence of states and events with the aim of making the goals true. States are modeled as sets of facts representing properties that change over time (called *fluents*). Events include both external events and internally generated actions. Rules modeling state transitions, called *causal laws*, based upon an underlying *causal*

theory, can destructively assert and retract fluents. Rules in the causal theory exploit an explicit representation of time. The language seems to be related to event calculus (introduced by Kowalski and Sergot in [58]) and to Action Languages [59] [60] [61] [62] although the latter are based on Answer Set Programming and thus theories admit multiple models, while an LPS theory has a single model. The LPS language includes reactive rules of the form *if antecedent then consequent* that transform LPS theories into agent programs. In fact, reactive rules link external events perceived by the agent and/or internal events generated within the agent itself, which occur in the antecedent of a reactive rule, to the LPS causal rules aimed to make the consequent true. AgentSpeak, DALI and LPS all feature reactive rules (called, in LPS, ECA rules). To manage concurrent events, requiring as a reaction actions that cannot reasonably occur together, LPS can adopt either time (to distinguish the two) or an integrity constraint to make them incompatible. In DALI, as events are time-stamped, they would be served in the order in which they arrived, however, it is possible to specify, as a user directive, a priority indicating which of the two should be considered first (e.g., referring to the example proposed in the LPS paper, better first eating and then sleeping). AgentSpeak and DALI actions' preconditions can assure that no incompatible actions are attempted. To model frame axioms, i.e., quoting from the LPS paper:

> *if a fact is true in a given state, then it continues to be true in a later state, unless it is terminated by an event (either an external event or action) that occurs between the two states.*

in LPS destructive state changes are adopted; in DALI instead, time-stamped past events can be removed by user directives (precisely, made no longer actual rather than physically removed). Concerning the LPS example of an sos issued in consequence of three consequent flashes, the reader may refer to the alarm example presented in Section 3, which is quite analogous. The capabilities of DALI for Complex Event Processing are described in [63, 64]. Let us consider the more extensive LPS example presented in the paper by Kowalski et al. appearing in this issue. To understand the code, note that, where the execution of LPS programs, and computation in LPS in general, fulfills in principle a background event-calculus-like causal theory, the LPS implementation relies on destructive state updates: fluents initiated by some event are added to the current state, whereas fluents that are terminated are vice versa deleted.

$initially\ lightOn.$
$observe\ switch\ from\ 1\ to\ 2.$
$observe\ switch\ from\ 3\ to\ 4.$
$lightOff\ if\ not\ lightOn.$
$switch\ initiates\ lightOn\ if\ lightOff.$
$switch\ terminates\ lightOn\ if\ lightOn.$

It is not clear whether the above LPS agent is meant to hypothesize the state of the external environment (precisely, the light being on or off) in consequence of observing switch events, or if the agent is acting itself to control the light source. The DALI formulation shown below works in both cases, where actions are in the former case fictitious (do nothing) and are just used to create past events to hypothesize the current environment state, while in the latter case actions actually switch the light on and off.

$initially\_lightOnE{:>}\ lightOnA.$
$switchE{:>}\ choose\_switch.$
$choose\_switch :-\ lightOnP,\ lightOffA.$
$choose\_switch :-\ lightOffP,\ lightOnA.$
$keep\ lightOnP\quad until\ lightOffA.$
$keep\ lightOffP\ \ until\ lightOnA.$

In case one is really interested in defining a fluent that makes it explicit in which interval a switch has been observed, this can be done as seen in Section 3.

In LPS, both the antecedent and consequent of a reactive rule can include conjunctions of timeless predicates, fluents, and events, possibly negated. This allows for effects similar to DALI's internal events but less explicit and without frequency for attempts. Unlike DALI and AgentSpeak, LPS does not allow specifying execution deadlines for reactive rules. Additionally, LPS does not address Multi-Agent systems as it lacks mechanisms for inter-agent communication.

The paper by M. Genesereth discusses Epilog [65], a Dynamic Logic Programming language, which is intended as a Knowledge Representation formalism for describing the world. It has often been used for describing legal actions, with the related/required actions and/or goals. Where in LPS, an agent behavior is formalized by means of behavioral constraints, and explicitly mentions time, Epilog programs are composed of operation rules (for short 'rules') of the form $action :: conditions ==> effects$. The action expression to the left of the double colon is called the head, the literals to the left of the arrow are conditions, and the literals to its right are effects. Operationally, if the conditions of a rule are true in any state, executing the action in the head requires executing the effects of the rule. The semantics of stratified (with respect to negation) Epilog programs is defined by their extension, which is the union of all datasets obtained from an initial dataset Δ via repeated application of all applicable rules, resulting in a unique extension for a given program and dataset. Epilog executes all applicable operation rules in parallel, applying all updates (both deletions and additions) to the dataset, thus completing a step and enabling the rules to fire again.

Both DALI and AgentSpeak can also execute all distinct activities in parallel using an enhanced interpreter. Epilog rules can model agent actions in response to external stimuli using Prolog-style rules called Views, although this concept is not further elaborated in the paper. An example provided in the paper is Tic Tac Toe, where two players (denoted by identifiers x and o) place a mark on a blank cell by retracting the cell fact with mark b and asserting the fact with the new mark, then passing control to the other player by retracting a fact and asserting a new one.

$$mark(M,N) :: control(Z) ==> cell(M,N,b) \,\&\, cell(M,N,Z)$$
$$mark(M,N) :: control(x) ==> control(x) \,\&\, +control(o)$$
$$mark(M,N) :: control(o) ==> control(o) \,\&\, control(x)$$

In AgentSpeak, these rules would look like:

$$+!mark(M,N) : control(Z) <- -cell(M,N,b) ; +cell(M,N,Z)$$
$$+!mark(M,N) : control(x) <- -control(x) ; +control(o)$$
$$+!mark(M,N) : control(o) <- -control(o) ; -control(x)$$

where however it would be easy to devise three agents, player x, player o and $table$; the players would require $table$ to assert their move, and the table would inform them if one wins (this via the *confirm* and *inform* FIPA primitive). Agent-related issues are hardly considered in Epilog, and inter-agent communication is not considered at all.

Overall, in our view the different approaches presented in this issue (DALI, AgentSpeak, LPS, Epilog) have some similar features concerning, e.g., reactive rules, but on many aspects they are complementary and present their own distinguished style of modeling problems. Thus, it is difficult to consider these approaches as in competition with each other. The features that characterize each approach can be profitably exploited in some specific application domains, as discussed by the authors themselves in the aforementioned references, and may be less suitable in other contexts. Notice, however, that DALI and AgentSpeak have practically proven to have wide applicability, even in industrial applications, while the others remain at the moment more theoretical than practical. Interesting future work may concern the investigation of how these approaches might be profitably integrated into a comprehensive framework so as to exploit in a synergistic way the features of each.

8. Conclusions

In this paper we have presented, discussed and compared Agent-Oriented logic languages, with particular attention to those, like AgentSpeak and DALI, that build upon Prolog's foundation to allow for the specification and implementation of dynamic systems. These languages offer significant advantages over traditional Prolog for this purpose. We have seen that AgentSpeak and DALI incorporate mechanisms for both reactive (responding to events) and proactive (taking initiative) agent behavior. Additionally, their semantics can be grounded in Prolog-like logic through a technique called Evolutionary Semantics. This approach translates agent programs with a limited form of assert/retract statements into standard Prolog, making them easier to reason about and verify. This not only simplifies the development process

but also allows for formal verification of agent behavior, a crucial aspect for ensuring robust and reliable systems. In conclusion, we would like to reiterate the importance of incorporating functions for the development of multi-agent systems in such languages. This paves the way for modeling a much wider range of practical applications. More importantly, it enables the specification of distributed logic-based agent systems. Languages such as DALI and AgentSpeak, which extend agent-oriented programming with MAS functions, can be further enhanced by adhering to communication protocols such as FIPA (or other ACLs). FIPA compliance provides interoperability among agent systems written in disparate languages, anticipating a future where logical programming forms the foundation of a unified network of intelligent agents spanning the entirety of the Internet. Collaborative problem solving, distributed information processing and the automation of complicated systems become immensely possible. As a result, the scope of logic programming is expanding exponentially and encompasses the broad field of the Internet.

References

[1] M. Wooldridge, An introduction to multiagent systems, John wiley & sons, 2009.

[2] A. Vozna, A. Monaldini, S. Costantini, P. d. Giovanni De Gasperis, A. Formisano, A. Rafanelli, Evolution of programming languages in agent systems, 2024. Submitted paper.

[3] J. Bloch, Effective java, Addison-Wesley Professional, 2017.

[4] D. Robinson, R. Ferrigno, J. Silge, D. Choi, Why is python growing so quickly?, Stack Overflow blog (2017).

[5] R. A. Brooks, Intelligence without reason, in: The artificial life route to artificial intelligence, Routledge, 2018, pp. 25–81.

[6] R. A. Brooks, Intelligence without representation, Artificial intelligence 47 (1991) 139–159.

[7] A. S. Rao, M. P. Georgeff, Modeling rational agents within a BDI-architecture, in: R. Fikes, E. Sandewall (Eds.), Proceedings of Knowledge Representation and Reasoning (KR&R-91), Morgan Kaufmann Publishers: San Mateo, CA, 1991, pp. 473–484.

[8] A. S. Rao, M. Georgeff, BDI Agents: from theory to practice, in: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, 1995, pp. 312–319.

[9] S. Costantini, P. Dell'Acqua, G. A. Lanzarone, Reflective agents in metalogic programming, in: A. Pettorossi (Ed.), Meta-Programming in Logic, 3rd International Workshop, META-92, Proceedings, volume 649 of *Lecture Notes in Computer Science*, Springer, 1992, pp. 135–147.

[10] R. A. Kowalski, F. Sadri, Towards a unified agent architecture that combines rationality with reactivity, in: D. Pedreschi, C. Zaniolo (Eds.), Logic in Databases, International Workshop LID'96, San Miniato, Italy, July 1-2, 1996, Proceedings, volume 1154 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 137–149.

[11] J. Barklund, P. Dell'Acqua, S. Costantini, G. A. Lanzarone, Multiple metareasoning agents for flexible query-answering systems, in: H. Christiansen, H. L. Larsen, T. Andreasen (Eds.), Flexible Query-Answering Systems, Proceedings of the 1996 Workshop, FQAS'96, Roskilde, Denmark, May 22-24, 1996, volume 62 of *Datalogiske Skrifter (Writings on Computer Science)*, Roskilde University, 1996, pp. 155–166.

[12] P. Dell'Acqua, F. Sadri, F. Toni, Combining introspection and communication with rationality and reactivity in agents, in: J. Dix, L. F. del Cerro, U. Furbach (Eds.), Logics in Artificial Intelligence, European Workshop, JELIA '98, Dagstuhl, Germany, October 12-15, 1998, Proceedings, volume 1489 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 17–32.

[13] J. Barklund, K. Boberg, P. Dell'Acqua, M. Veanes, Meta-programming with theory systems, in: K. Apt, F. Turini (Eds.), Meta-Logics and Logic Programming, The MIT Press, Cambridge, Mass., 1995, pp. 195–224.

[14] R. A. Kowalski, F. Sadri, Towards a unified agent architecture that combines rationality with reactivity, in: Proc. International Workshop on Logic in Databases, LNCS 1154, Springer-Verlag, Berlin, 1996.

[15] P. Dell'Acqua, F. Sadri, F. Toni, Combining introspection and communication with rationality and reactivity in agents, in: J. Dix, F. D. Cerro, U. Furbach (Eds.), Logics in Artificial Intelligence, LNCS 1489, Springer-Verlag, Berlin, 1998.

[16] P. Dell'Acqua, F. Sadri, F. Toni, Communicating agents, in: Proc. International Workshop on Multi-Agent Systems in Logic Programming, in conjunction with ICLP'99, Las Cruces, New Mexico, 1999.

[17] S. Costantini, Meta-reasoning: a Survey, in: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II, volume 2408 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 253–288.

[18] S. Costantini, Towards active logic programming, in: A. Brogi, P. Hill (Eds.), Proc. of 2nd International Workshop on Component-based Software Development in Computational Logic (COCL'99), PLI'99, http://www.di.unipi.it/ brogi/ ResearchActivity/COCL99/ proceedings/index.html, Paris, France, 1999.

[19] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, A. Ricci, A survey of programming languages and platforms for multi-agent systems, Informatica (Slovenia) 30 (2006) 33–44.

[20] A. Garro, M. Mühlhäuser, A. Tundis, M. Baldoni, C. Baroglio, F. Bergenti, P. Torroni, Intelligent agents: Multi-agent systems, in: S. Ranganathan, M. Gribskov, K. Nakai, C. Schönbach (Eds.), Encyclopedia of Bioinformatics and Computational Biology - Volume 1, Elsevier, 2019, pp. 315–320. doi:10.1016/b978-0-12-809633-8.20328-2.

[21] R. Calegari, G. Ciatto, V. Mascardi, A. Omicini, Logic-based technologies for multi-agent systems: a systematic literature review, Auton. Agents Multi Agent Syst. 35 (2021) 1. doi:10.1007/s10458-020-09478-3.

[22] S. Costantini, A. Tocchio, A logic programming language for multi-agent systems, in: S. Flesca, S. Greco, N. Leone, G. Ianni (Eds.), Logics in Artificial Intelligence, European Conference, JELIA 2002, Proceedings, volume 2424 of *Lecture Notes in Computer Science*, Springer, 2002.

[23] S. Costantini, A. Tocchio, The DALI logic programming agent-oriented language, in: J. J. Alferes, J. A. Leite (Eds.), Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Proceedings, volume 3229 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 685–688.

[24] R. H. Bordini, J. F. Hübner, Semantics for the jason variant of agentspeak (plan failure and some internal actions), in: H. Coelho, R. Studer, M. J. Wooldridge (Eds.), ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings, volume 215 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2010, pp. 635–640. doi:10.3233/978-1-60750-606-5-635.

[25] F. F. Ingrand, M. P. Georgeff, A. S. Rao, An architecture for real-time reasoning and system control, IEEE expert 7 (1992) 34–44.

[26] R. H. Bordini, M. Dastani, J. Dix, A. E. F. Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer, 2005.

[27] R. H. Bordini, M. Dastani, J. Dix, A. E. F. Seghrouchni (Eds.), Multi-Agent Programming, Languages, Tools and Applications, Springer, 2009. URL: https://doi.org/10.1007/978-0-387-89299-3. doi:10.1007/978-0-387-89299-3.

[28] A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: Agents Breaking Away, 7th European Works. on Modelling Autonomous Agents in a Multi-Agent World, Proceedings, volume 1038 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 42–55.

[29] R. H. Bordini, J. F. Hübner, Bdi agent programming in agentspeak using jason, in: International workshop on computational logic in multi-agent systems, Springer, 2005, pp. 143–164.

[30] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason, John Wiley & Sons, 2007.

[31] R. W. Collier, S. Russell, D. Lillis, Reflecting on agent programming with agentspeak (l), in: PRIMA 2015: Principles and Practice of Multi-Agent Systems: 18th International Conference, Bertinoro, Italy, October 26-30, 2015, Proceedings 13, Springer, 2015, pp. 351–366.

[32] V. Bevar, S. Costantini, A. Tocchio, G. D. Gasperis, A multi-agent system for industrial fault detection and repair, in: Y. Demazeau, J. P. Müller, J. M. C. Rodríguez, J. B. Pérez (Eds.), Advances on Practical Applications of Agents and Multi-Agent Systems - Proc. of PAAMS 2012, volume 155 of *Advances in Soft Computing*, Springer, 2012, pp. 47–55. Related Demo paper "Demonstrator of a Multi-Agent System for Industrial Fault Detection and Repair", pages 237-240 of same volume.

[33] S. Costantini, G. De Gasperis, G. Nazzicone, DALI for cognitive robotics: Principles and prototype implementation, in: Y. Lierler, W. Taha (Eds.), Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Proceedings, volume 10137 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 152–162.

[34] S. Costantini, G. De Gasperis, Dynamic goal decomposition and planning in MAS for highly changing environments, in: Proceedings of the 33rd Italian Conference on Computational Logic, volume 2214 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 40–54.

[35] G. De Gasperis, S. Costantini, G. Nazzicone, Dali multi agent systems framework, doi 10.5281/zenodo.11042, DALI GitHub Software Repository, 2014. DALI: http://github.com/AAAI-DISIM-UnivAQ/DALI.

[36] V. Lifschitz, Answer set planning, in: D. D. Schreye (Ed.), Proc. of ICLP '99 Conference, MIT Press, Cambridge, Ma, 1999, pp. 23–37. Invited talk.

[37] S. Costantini, A. Tocchio, About declarative semantics of logic-based agent languages, in: Declarative Agent Languages and Technologies III, Third Intl. Works., DALT 2005, Selected and Revised Papers, volume 3904 of *LNAI*, Springer, 2006, pp. 106–123.

[38] J. W. Lloyd, Foundations of Logic Programming, Second Edition, Springer-Verlag, Berlin, 1987.

[39] K. R. Apt, Logic programming, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier and MIT Press, 1990, pp. 493–574. doi:10.1016/b978-0-444-88074-1.50015-9.

[40] R. Bordini, M. Fisher, W. Visser, M. Wooldridge, Verifying multi-agent programs by model checking, Autonomous Agents and Multi-Agent Systems 12 (2006) 239–256.

[41] A. Lomuscio, H. Qu, F. Raimondi, MCMAS: an open-source model checker for the verification of multi-agent systems, Int. J. Softw. Tools Technol. Transf. 19 (2017) 9–30.

[42] A. Ferrando, L. A. Dennis, D. Ancona, M. Fisher, V. Mascardi, Verifying and validating autonomous systems: Towards an integrated approach, in: C. Colombo, M. Leucker (Eds.), Runtime Verification - 18th International Conference, RV 2018, Proceedings, volume 11237 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 263–281. doi:10.1007/978-3-030-03769-7_15.

[43] A. Ferrando, M. Winikoff, S. Cranefield, F. Dignum, V. Mascardi, On enactability of agent interaction protocols: Towards a unified approach, in: L. A. Dennis, R. H. Bordini, Y. Lespérance (Eds.), Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Revised Selected Papers, volume 12058 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 43–64. doi:10.1007/978-3-030-51417-4_3.

[44] N. Hanna, D. Richards, Speech act theory as an evaluation tool for human–agent communication, Algorithms 12 (2019) 79.

[45] B. Smith, Towards a history of speech act theory, Speech acts, meanings and intentions. Critical approaches to the philosophy of John R. Searle (1990) 29–61.

[46] R. Vieira, Á. F. Moreira, M. Wooldridge, R. H. Bordini, On the formal semantics of speech-act based communication in an agent-oriented programming language, Journal of Artificial Intelligence Research 29 (2007) 221–267.

[47] G. De Giacomo, Y. Lespérance, L. H. J., Congolog, a concurrent programming language based on the situation calculus, Artificial Intelligence (2000) 109–169.

[48] K. V. Hindriks, F. de Boer, W. van der Hoek, J.-J. Meyer, Agent programming in 3apl, Autonomous Agents and Multi-Agent Systems 2 (1999) 357–401.

[49] K. Hindriks, F. de Boer, W. van der Hoek, J. J. Meyer, A formal architecture for the 3apl programming language, in: Proceedings of the First International Conference of B and Z Users, Springer Verlag, Berlin, 2000.

[50] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, R. Ross, Heterogenous Active Agents, MIT-Press, 2000. 580 pages.
[51] M. Fisher, A survey of concurrent METATEM – the language and its applications, in: Proceedings of First International Conference on Temporal Logic (ICTL), LNCS 827, Springer Verlag, Berlin, 1994.
[52] M. Mulder, J. Treur, M. Fisher, Agent modelling in concurrent METATEM and DESIRE, in: Intelligent Agents IV, LNAI, Springer Verlag, Berlin, 1998.
[53] R. A. Kowalski, F. Sadri, M. Calejo, J. A. Dávila, Combining logic programming and imperative programming in LPS, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, volume 13900 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 210–223. doi:10.1007/978-3-031-35254-6_17.
[54] M. R. Genesereth, Dynamic logic programming, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, volume 13900 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 197–209. doi:10.1007/978-3-031-35254-6_16.
[55] R. Kowalski, F. Sadri, Reactive computing as model generation, New Generation Computing 33 (2015) 33–67.
[56] R. A. Kowalski, F. Sadri, Programming in logic without logic programming, Theory Pract. Log. Program. 16 (2016) 269–295. URL: https://doi.org/10.1017/S1471068416000041. doi:10.1017/S1471068416000041.
[57] R. A. Kowalski, F. Sadri, M. Calejo, How to do it with lps (logic-based production system)., in: RuleML+ RR (Supplement), 2017.
[58] R. Kowalski, M. Sergot, A logic-based calculus of events, New Generation Computing 4 (1986) 67–95.
[59] C. Barai, M. Gelfond, Logic programming and reasoning about actions, in: Foundations of Artificial Intelligence, volume 1, Elsevier, 2005, pp. 389–426.
[60] J. Blount, M. Gelfond, Reasoning about the intentions of agents, in: Logic Programs, Norms and Action: Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday, Springer, 2012, pp. 147–171.
[61] D. Inclezan, M. Gelfond, Modular action language, Theory and Practice of Logic Programming 16 (2016) 189–235.
[62] C. Baral, G. Gelfond, E. Pontelli, T. C. Son, An action language for multi-agent domains, Artificial Intelligence 302 (2022) 103601.
[63] S. Costantini, P. Dell'Acqua, A. Tocchio, Expressing preferences declaratively in logic-based agent languages, in: Proceedings of Commonsense'07, the 8th International Symposium on Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium Series, 2007. A special event in honor of John McCarthy.
[64] S. Costantini, ACE: a flexible environment for complex event processing in logical agents, in: L. B. Matteo Baldoni, M. Dastani (Eds.), Engineering Multi-Agent Systems, Third International Workshop, EMAS 2015, Revised Selected Papers, volume 9318 of *Lecture Notes in Computer Science*, Springer, 2015.
[65] M. Genesereth, Epilog for javascript, 2013.