Computational Logic in an Object-Oriented World

Robert Kowalski

Imperial College London
rak@doc.ic.ac.uk

Abstract Logic and object-orientation (OO) are competing ways of looking at the world. Both view the world in terms of individuals. But logic focuses on the relationships between individuals, and OO focuses on the use of hierarchical classes of individuals to structure information and procedures. In this paper, I investigate the similarities and differences between OO and abductive logic programming multi-agent systems (ALP systems) and argue that ALP systems can combine the advantages of logic with the main benefits of OO.

In ALP systems, relationships between individuals are contained in a shared semantic structure and agents interact both with one another and with the environment by performing observations and actions. In OO systems, on the other hand, relationships are associated with objects and are represented by attribute-value pairs. Interaction between objects is performed by sending and receiving messages.

I argue that logic can be reconciled with OO by combining the hierarchical, modular structuring of information and procedures by means of objects/agents, with a shared semantic structure, to store relationships among objects/individuals, accessed by observations and actions instead of by message passing.

Keywords: object-orientation, ALP systems, multi-agent systems, logic programming, Linda

1 Introduction

There was a time in the 1980s when it seemed that Computational Logic (CL) might become the dominant paradigm in Computing. By combining the declarative semantics of logic with the computational interpretation of its proof procedures, it could be applied to virtually all areas of Computing, including program specification, programming, databases, and knowledge representation in Artificial Intelligence.

But today it is Object-Orientation (OO), not Logic, that dominates every aspect of Computing – from modelling the system environment, through specifying system requirements, to designing and implementing the software and hardware. Like CL, OO owes much of its attraction, not only to its computational properties, but also to its way of thinking about the world. If these attractions have real substance, then they potentially undermine not only CL's place inside Computing, but also its way of modelling and reasoning about the world outside Computing.

The aim of this paper is to try to understand what makes OO so attractive and to determine whether these attractions can be reconciled with CL, both in Computing and in the wider world. I will argue that logic-based multi-agent systems can combine the advantages of CL with the main benefits of OO.

I will illustrate my argument by using abductive logic programming (ALP) multi-agent systems [1]. However, most of the argument applies to more general logic-based multi-agent systems, and even to heterogeneous systems that use different programming languages, provided their external interfaces can be viewed in logical terms.

ALP multi-agent systems (ALP systems, in short) are semantic structures, consisting of individuals and relationships, as in the conventional semantics of classical logic. However, in ALP systems, these structures can change state, in the same way that the real world changes state, destructively, without remembering its past. Some individuals in the structure are agents, which interact with the world, by observing the world and by performing actions on the world. Other individuals passively undergo changes performed by agents, and still other individuals, like numbers, are immutable and timeless.

ALP agents, which are individuals in the ALP semantic structure, also have an internal, syntactic structure, consisting of goals and beliefs, which they use to interact with the world. Their beliefs are represented by logic programs, and their goals are represented by integrity constraints. Their observations and actions are represented by abducible (undefined) predicates.

I argue that such logic-based multi-agent systems share many of the attractions of OO systems. In particular, they share with objects the view that the world consists of individuals, some of which (objects or agents) interact with other individuals and change the state of the world. However, whereas in OO systems relationships among individuals are associated with objects and are represented as attribute-value pairs, in ALP systems relationships belong to the semantic structure of the world.

Both agents in ALP systems and objects in OO systems encapsulate their methods for interacting with the world, hiding their implementation details from other agents and objects. Both agents and objects can inherit their methods from more general classes of agents or objects. Whereas objects use methods implemented in conventional, imperative programming languages, ALP agents use methods implemented by means of goals and beliefs in logical form. The methods used by ALP agents have both a procedural behaviour, as well as a declarative semantics. In the declarative semantics, the goals and beliefs of an agent have a truth value in the semantic structure that is the ALP system as a whole. Normally, beliefs that are true and goals that can be made true are more useful to an agent than ones that are false[1].

Both ALP systems and OO systems share a local notion of change, in which changes can take place in different parts of the world locally, concurrently and independently. This local notion of change contrasts with the global notion that is prevalent in most logical treatments, including the possible world semantics of modal logic and the situation calculus. The global notion of change is useful for theoretical

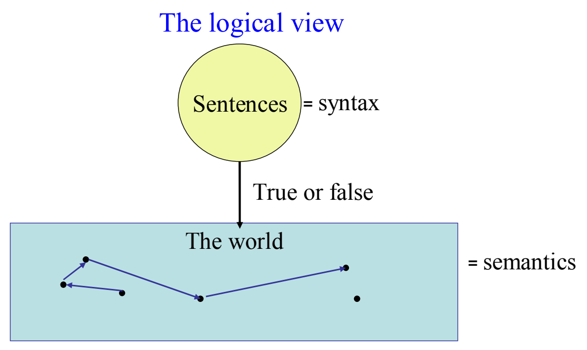
[1] A false belief can be more useful than a true belief, if the truth is too complicated to use in practice.

purposes, but the local notion is more useful both as a model of the real world and as a model for constructing artificial worlds.

ALP agent systems differ from OO systems in one other important respect: Whereas objects interact by sending and receiving messages, agents interact by observing and performing actions on the shared semantic structure. This semantic structure acts as a shared environment, similar to the blackboard in a blackboard system [8] and to the tuple-space in a Linda programming environment [2]. In the same way that Linda processes can be implemented in different and heterogeneous programming languages, the methods used by ALP agents can also be implemented in other programming languages, provided their externally observed behaviour can be viewed in logical terms.

In the remainder of the paper, I will first introduce ALP systems in greater detail and then distinguish between the semantic and syntactic views of OO systems. I will then compare OO systems and ALP systems by investigating how each kind of system can be simulated by the other. The directness of these simulations is the basis for the comparison of the two approaches. The simulations are informal and should be viewed more as illustrations than as outlines of formal theorems.

2 The Logical Way of Looking at the World



- The world gives meaning to sentences.
- Individuals are named by terms.
- Relations are named by predicate symbols.
- Sentences of logic correspond to declarative sentences of natural language.

Figure 1. The relationship between syntax and semantics.

In logic there is a clear distinction between syntax and semantics. Syntax is concerned with the grammatical form of sentences and with the inference rules that derive conclusion sentences from assumption sentences. Semantics is concerned with the individuals and relationships that give sentences their meaning. The relationship between syntax and semantics is pictured roughly in figure 1.

The distinction between atomic sentences and the semantic relationships to which they refer is normally formalised by defining an interpretation function, which interprets constant symbols as naming individuals and predicate symbols as naming relations. However, it is often convenient to blur the distinction by restricting attention to *Herbrand interpretations*, in which the semantic structure is identified with the set of all atomic sentences that are true in the structure. However, the use of Herbrand interpretations can sometimes lead to confusion, as in the case where a set of atomic sentences can be considered both semantically as a Herbrand interpretation and syntactically as a set of sentences. Sometimes, to avoid confusion, atomic sentences understood as semantically as relationships are also called *facts*.

For notational convenience, we shall restrict our attention to Herbrand interpretations in the remainder of the paper. However, note that, even viewing Herbrand interpretations as syntactic representations, there is an important sense in which they differ from other syntactic representations. Other syntactic representations can employ quantifiers and logical connectives, which generalize, abstract and compress many atomic sentences into a smaller number of sentences, from which other sentences, including the atomic sentences, can be derived.

2.1 ALP Agents

Traditional logic is often accused by its critics of being too concerned with static states of affairs and of being closed to changes in the world. The first of these criticisms has been addressed in various ways, either by making the semantics more dynamic, as in the possible worlds semantics of modal logic, or by making the syntax more expressive by reifying situations or events, as in the situation or event calculus.

The second of these criticisms has been addressed by embedding logic in the thinking component of the observation-thought-decision-action cycle of an intelligent agent, for example as in ALP agents, pictured in figure 2.

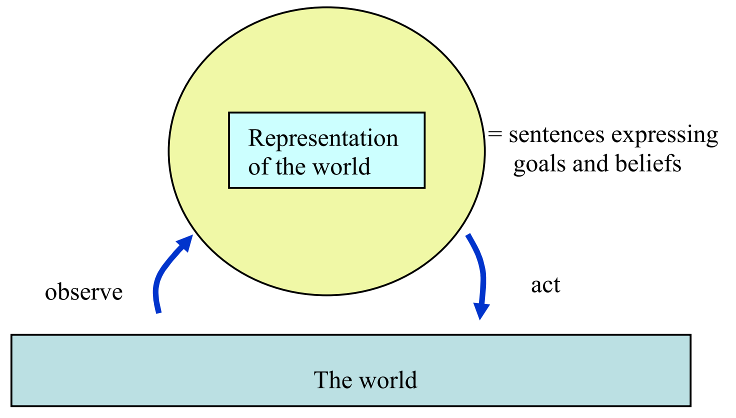

Representation of the world
= sentences expressing goals and beliefs
observe
act
The world

Figure 2. Logic as the thinking component of an intelligent agent

In ALP agents, beliefs are represented by logic programs and goals are represented by integrity constraints. Integrity constraints are used to represent a variety of kinds of goals, including maintenance goals, prohibitions, and condition-action rules. Abducible predicates, which are not defined by logic programs, but are restricted by the integrity constraints, are used to represent observations and actions.

ALP agents implement *reactive* behaviour, initiated by the agent's observations, using forward reasoning to trigger maintenance goals and to derive achievement goals. They also implement *proactive* behaviour, initiated by achievement goals, using backward reasoning to reduce goals to sub-goals and to derive action sub-goals. In addition to reactive and proactive thinking, ALP agents can also perform *pre-active* thinking [20], using forward reasoning to simulate candidate actions, to derive their likely consequences, to help in choosing between them.

2.2 An ALP Agent on the London Underground

Passengers on the London underground have a variety of goals – getting to work, getting back home, going out shopping or visiting the tourist attractions. In addition, most, law-biding passengers are also concerned about safety. This concern can be represented by goals in logical form, which might include the (simplified) goal:

If there is an emergency then I get help.

To recognize when there is an emergency and to find a way to get help, a passenger can use beliefs[2] in logic programming form:

I get help if I alert the driver.
I alert the driver if I press the alarm signal button.

There is an emergency if there is a fire.
There is an emergency if one person attacks another.
There is an emergency if someone becomes seriously ill.
There is an emergency if there is an accident.

The beliefs about getting help are declarative sentences, which may be true or false about the effect of actions on the state of the world. The beliefs about emergencies are also declarative sentences, but they are simply true by definition, because the concept of emergency is an abstraction without a direct interpretation in concrete experience.

In ALP, beliefs can be used to reason forwards or backwards. Forward reasoning is useful for deriving consequences of observations and candidate actions. Backward reasoning is useful for reducing goals to sub-goals. A combination of forward and backward reasoning in the London underground example is illustrated in figure 3.

[2] For simplicity, this representation of goal and beliefs ignores the element of time.

The mental activity of an ALP agent is encapsulated in the agent, hidden from an observer, who can see only the agent's input-output behaviour. In the case of the London underground passenger, this behaviour has the logical form:

If there is a fire, then the passenger presses the alarm signal button.

As far as the observer is concerned, this externally visible, logical form of the passenger's behaviour could be implemented in any other mental representation or programming language.

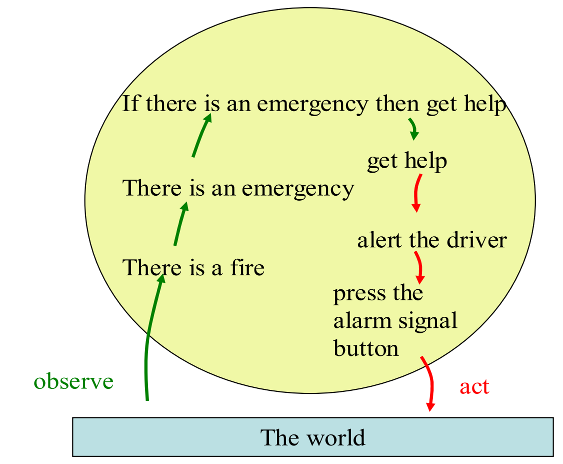


Figure 3. ALP combines forward and backward reasoning

2.3 ALP Agent Systems

Similarly to the way that agents interact with other individuals in real life, ALP agents interact with other individuals embedded in a shared environment. This environment is a semantic structure consisting of individuals and relationships. Some of the individuals in the environment are agents of change, while others undergo changes only passively. To a first approximation, we can think of an ALP environment as a relational database, which changes destructively as the result of agents' actions.

The environment that ALP agents share is a dynamic structure, in which relationships come and go as the result of actions, which occur locally, concurrently and independently of other actions. Because this environment is a semantic structure, relationships can appear and disappear destructively, without the environment having to remember the past. In the London underground example, the observations and actions of the passenger, train driver and fire department agents are illustrated in figure 4. Instead of standing apart and, as it were, above the world, as pictured in figures 1-3, the agents are embodied within it. Their actions change the shared environment by adding and deleting facts (or relationships):

> The passenger's action of pressing the alarm signal button
> *deletes* the fact that the alarm is off and
> *adds* the fact that the alarm is on.
>
> The driver's action of calling the fire department
> *adds* the fact that the fire department has been called.
>
> The fire department's action of putting out the fire
> *deletes* the fact that there is a fire in the train.

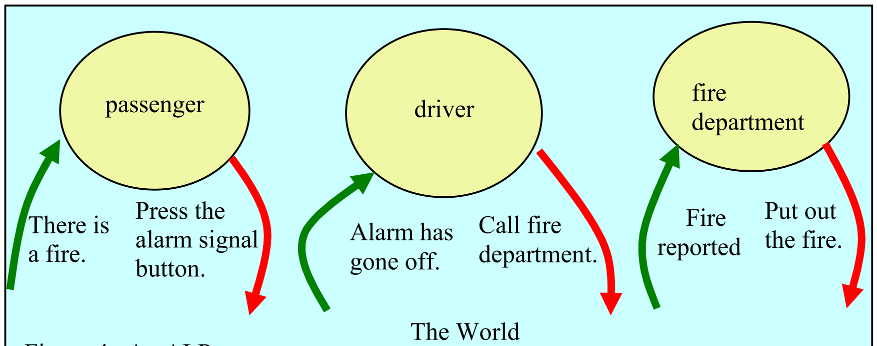


Figure 4. An ALP system.

The driver's action of calling the fire department can be viewed as sending the fire department a message, in the form of a fact that is stored in the shared environment. The fire department observes the message and, if it chooses, may delete it from the environment. Other agents may be able to observe the message, as long as it remains in the environment, provided they can access that part of the environment.

Notice that, just as in the case of a single agent, an observer can see only the agents' external behaviour. In this case also, that behaviour has a logical form:

> If there is a fire, then the passenger presses the alarm signal button.
> If the alarm has gone off, then the driver calls the fire department.
> If a fire is reported, then the fire department puts out the fire.

These implications can be combined with sentences describing the effect of the agents' actions on the world:

> If a person presses the alarm signal button, then the alarm goes off.
> If a person calls the fire department, then a fire is reported.

to derive the input-output behaviour of the combined system as a whole:

> If there is a fire, then the fire department puts out the fire.

3 Object-Oriented Systems

Despite the dominant position of OO in Computing, there seems to be no clear definition or consensus about its fundamental concepts. One recent attempt to do so [21] identifies inheritance, object, class, encapsulation, method, message passing, polymorphism, and abstraction, in that order, as its most frequently cited features. However, the relative importance of these concepts and their precise meaning differs significantly from one OO language to another. This makes comparison with logic very difficult and prone to error. Therefore, the claims and comparisons made in this paper need to be judged and qualified accordingly.

Nonetheless, viewed in terms of the concepts identified in [21], the argument of this paper can be simply stated as claiming that all of these concepts are either already a feature of ALP systems (and other, similar logic-based multi-agent systems) or can readily be incorporated in them, with the exception of *message-passing*.

OO shares with logic the view that the world consists of individuals, some of which (objects or agents) interact with other individuals and change the state of the world. In OO, objects interact with one another by sending and receiving messages, using encapsulated methods, which are hidden from external observers, and which are acquired from more general classes of objects, organised in hierarchies.

Whereas ALP agents use goals and beliefs to regulate their behaviour, objects use methods that are typically implemented by means of imperative programming language constructs. An object-oriented system corresponding to the multi-agent system of figure 4 is pictured in figure 5.

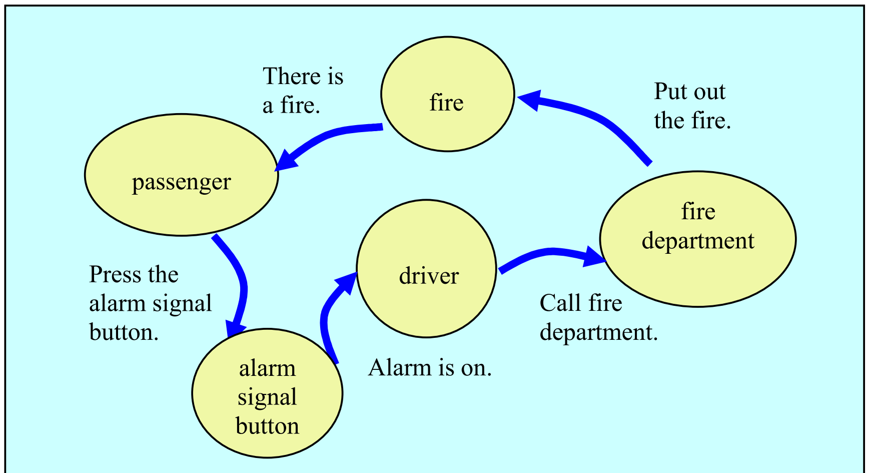


Figure 5. An object-oriented system corresponding to the multi-agent system of figure 4.

Both ALP systems and OO systems can be viewed as semantic structures, in which the world is composed of individuals that interact with one another and change state. However, there are important differences between them:

1. *The treatment of individuals.* In ALP systems, a distinction is made between agents, which are active, and other individuals, which are passive. In OO systems, both kinds of individuals are treated as objects.
2. *The treatment of attributes and relationships.* In the semantic structures of logic, individuals have externally observable attributes and relationships with other individuals. Attributes are treated technically as a special case of relationships.

   In OO systems, relationships between objects are treated as attributes of objects. Either one of the objects in a relationship has to be treated as its "owner", or the relationship needs to be represented redundantly among several "owners".
3. *The way of interacting with the world.* ALP agents interact with the world by observing the current state of the world and by performing actions to change it. A relationship between several individuals can be accessed in a single observation; and a single action can change the states of several relationships.

   Objects in OO systems, on the other hand, interact with one another directly by sending and receiving messages. But the concept of "message" is not defined. In many - perhaps most - cases, messages are used to request help from other objects in solving sub-goals. In other cases, messages are used to send information (including solutions of sub-goals) from one object to another. But in the general case, in many OO languages, messages can be used for any arbitrary purpose.

3.1 Object-oriented Systems as Syntactic Structures

The relationship between logic and objects can be viewed in both semantic and syntactic terms. However, it is the syntactic structuring of information and methods into encapsulated hierarchies of classes of objects that is perhaps the most important reason for the practical success of OO in Computing.

In ALP systems, information and methods are syntactically formulated by means of goals and beliefs in logical form. In OO systems, methods are typically implemented in an imperative language. In both cases, internal processing is encapsulated, hidden from other agents or objects, and performed by manipulating sentences in a formal language.

In logic, there is a well understood relationship between syntax and semantics, in which declarative sentences are either true or false. In ALP agents, declarative sentences representing an agent's goals and beliefs are similarly true or false in the semantic structure in which the agent is embedded.

In OO systems, where methods are implemented in imperative languages, there is no obvious relationship between the syntax of an object's methods and the semantic structure of the OO system as a whole. In part, this is because purely imperative languages do not have a simple truth-theoretic semantics; and, in part, because messages do not have a well defined intuitive interpretation.

3.2 Natural Language and Object-orientation

We can better understand the nature of OO and logical syntax by comparing them both with natural language. Comparing logic with natural language, an important difference is that sets of sentences in logic are unstructured and can be written in any order, without affecting their meaning. But in natural language, the way in which sentences are grouped together and the order in which they are presented affects both their meaning and their intelligibility.

In contrast with logic, but like natural language, OO is also concerned with the structuring of sentences. It structures sentences by associating them with the objects that the sentences are about. Natural languages, like English, employ a similar form of object-orientation by using grammatical structures in which the beginning of a sentence indicates a *topic* and the rest of the sentence is a *comment* about the topic. This kind of structure often coincides with, but is not limited to, the grammatical structuring of sentences into *subjects*[3] and *predicates*.

Consider, for example, the pair of English sentences [3, p. 130]:

> The prime minister stepped off the plane.
> Journalists immediately surrounded her.

Both sentences are formulated in the active voice, which conforms to the guidelines for good writing style advocated in most manuals of English.

The two sentences refer to three individuals/objects, the prime minister (referred to as "her" in the second sentence), journalists and the plane. The prime minister is the only object in common between the two sentences. So, the prime minister is the object that groups the two sentences together. However, the topic changes from the prime minister in the first sentence to the journalists in the second.

Now consider the following logically equivalent pair of sentences:

> The prime minister stepped off the plane.
> She was immediately surrounded by journalists.

Here the two sentences have the same topic, which is the individual/object they have in common. However, the second sentence is now expressed in the passive voice.

Despite the fact that using the passive voice goes against the standard guidelines of good writing style, most people find the second pair sentences easier to understand. This can be interpreted as suggesting that people have a strong preference for organising their thoughts in object-oriented form, which is even stronger than their preference for the active over the passive voice.

However, OO is not the only way of structuring sentences. Both linguists and proponents of good writing style have discovered a more general way, which includes OO as a special case. As Joseph Williams [4] argues:

[3] In this analogy between objects and topics, objects are more like the grammatical subjects of sentences than they are like the grammatical objects of sentences.

Whenever possible, express at the beginning of a sentence ideas already stated, referred to, implied, safely assumed, familiar, predictable, less important, readily accessible.

Express at the end of a sentence the least predictable. The newest, the most important, the most significant information, the information you almost certainly want to emphasize.

This more general way of structuring sentences also includes the use of logical form to make sets of sentences easier to understand. For example:

A if B.	or	D.
B if C.		If D then C.
C if D.		If C then B.
D.		If B then A.

3.3 Classes in Object-Oriented Systems Correspond to Sorts in Logic

Perhaps the most important practical feature of OO systems is the way in which objects acquire their methods from more general classes of objects. For example, an individual passenger on the underground can obtain its methods for dealing with fires from the more general class of all humans, and still other methods from the class of all animals.

Thus, classes can be organised in taxonomic hierarchies. Objects acquire their methods from the classes of which they are instances. Similarly sub-classes can inherit their methods from super-classes higher in the hierarchy, possibly adding methods of their own.

However, classes and class hierarchies are neither unique nor original features of OO systems. Classes correspond to types or sorts in many-sorted logics, and hierarchies of classes correspond to hierarchies of sorts in order-sorted logics.

Sorts and hierarchies of sorts can be (and have been) incorporated into logic programming in many different ways. Perhaps the simplest and most obvious way is by employing explicit sort predicates, such as Passenger(X) or Human(X), in the conditions of clauses, together with clauses defining sort hierarchies and instances, such as:

Human(X) if Passenger(X)
Passenger(john).

However, even unsorted logic programs already have a weak, implicit hierarchical sort structure in the structure of terms. A term f(X), where f is a function symbol, can be regarded as having sort f( ). The term f(g(X)), which has sort f(g( )) is a sub-sort of f( ), and the term f(g(a)), where a is a constant symbol is an instance of sort f(g()). Two terms that differ only in the names of variables, such as f(g(X)) and f(g(Y)) have the same sort. Simple variables, such as X and Y, have the universal sort. Both explicitly and implicitly sorted logic programs enjoy the benefits of inheritance[4].

Although sorts and inheritance are already features of many systems of logic, OO goes further by grouping sentences into classes. Sentences that are about several classes, such as the methods for humans dealing with fire, have to be associated either with only one of the classes or they have to be associated with several classes redundantly.[5]

3.4 Object-Oriented Logic Programming

We can better understand the relationship between OO systems and ALP systems if we see what is needed to transform one kind of system into the other. First, we will show how, under certain restrictions, logic programs can be transformed into OO systems. Later, we will show how to extend this transformation to ALP systems, and then we will show how to transform OO systems into ALP systems. In each of these cases, the OO system is an idealized system, corresponding to no specific OO language in particular. Thus the claims made about these transformations need to be qualified by this limitation.

OO grouping of sentences into classes can be applied to any language that has an explicit or implicit class structure, including sentences written in formal logic. However, as we have just observed, an arbitrary sentence can be about many different individuals or classes, making it hard to choose a single individual or class to associate with the sentence.

But it is easier to choose a class/sort for clauses in logic programs that define input-output predicates. For such programs, it can be natural to nominate one of the input arguments of the conclusion of a clause (or, more precisely, the sort of that argument) to serve as the "owner" of the clause. The different instances of the nominated input argument behave as sub-classes and objects, which use the clause as a method to reduce goals to sub-goals.

Whereas arbitrary messages in OO systems may not have a well-defined intuitive interpretation, messages in OO systems that implement input-output logic programs either send requests to solve goals and sub-goals or send back solutions. An object responds to a message requesting the solution of a goal by using a clause to reduce the goal to sub-goals. The object sends messages, in turn, requesting the solution of the

[4] Inheritance can be inhibited by the use of abnormality predicates. For example, the clauses Fly(X) if Bird(X) and not Abnormal(X), Bird(X) if Penguin(X), Walk(X) if Penguin(X), Swim(X) if Penguin(X), Abnormal(X) if Penguin(X) prevent penguins from inheriting the property of flying from the more general class of all birds.

[5] The problem of choosing a class or class hierarchy to contain a given sentence is similar to the problem of choosing a folder to store a file. Search engines like Google make it possible to store information in one structure but to access it without reference to the place it is stored. It also makes it possible to store information in an unstructured way, without any penalty in accessing it. Email clients offer similar, but more limited facilities to order emails by such different attributes as sender, date, subject, size, etc., without needing to duplicate them.

sub-goals, to the objects that are the owners of the sub-goals. When an object solves a goal or sub-goal, it sends the solution back to the object that requested the solution.

More formally, let a logic program contain the clause:

$$P_0(o_0, t_{01}, \ldots, t_{0m0}) \text{ if } P_1(o_1, t_{11}, \ldots, t_{1m1}) \text{ and } \ldots \text{ and } P_n(o_n, t_{n1}, \ldots, t_{nmn})$$

where, without loss of generality, the sort of the first argument of each predicate is selected as the owner of the clause. We also call that argument the "owner argument". Assume also that each such owner argument o_i is an input argument in the sense that at the time the predicate is invoked, as a goal or sub-goal for solution, the argument o_i is instantiated to some object (variable-free term).

Assume for simplicity that the sub-goals in the body of the clause are executed, Prolog-fashion, in the order in which they are written. Then the use of the clause to solve a goal of the form $P_0(o'_0, t'_{01}, \ldots, t'_{0m0})$, where o'_0 is a fully instantiated instance of o_0, is simulated by some sender object o sending the goal in a message to the receiver object o'_0 and by the receiver object:

0. matching the goal with the head of the clause,
   obtaining some most general unifying substitution θ_0

1. sending a message to object $o_1 \theta_0$ to solve the sub-goal
      $P_1(o_1, t_{11}, \ldots, t_{1m1}) \theta_0$
   receiving a message back from $o_1 \theta_0$ reporting that the sub-goal
      $P_1(o_1, t_{11}, \ldots, t_{1m1}) \theta_0$ has been solved with substitution θ_1
   ……

n. sending a message to object $o_n \theta_0\theta_1\ldots \theta_{n-1}$ to solve the goal
      $P_n(o_n, t_{n1}, \ldots, t_{nmn}) \theta_0\theta_1\ldots \theta_{n-1}$
   receiving a message back from $o_n \theta_0\theta_1\ldots \theta_{n-1}$ that the goal
      $P_n(o_n, t_{n1}, \ldots, t_{nmn}) \theta_0\theta_1\ldots \theta_{n-1}$ has been solved with substitution θ_n

n+1. sending a message back to the sender object o that the goal
      $P_0(o_0, t_{01}, \ldots, t_{0m0}) \theta_0$ has been solved with substitution $\theta_0\theta_1\ldots \theta_{n-1} \theta_n$.

Notice that objects o_i and o_j need not be distinct. The special case of an object sending a message to itself can be short circuited by the object simply solving the sub-goal locally itself.

If n=0, then the clause represents a relationship between the owner object and other individuals. The relationship is represented only once, associated with the owner object. All such relationships, like all other clauses, are encapsulated and can be accessed by other objects only by sending and receiving messages.

For example, the atomic clauses:

Father(john, bill)
Father(john, jill)
Mother(mary, bill)
Mother(mary, jill)

would be encapsulated within the john and mary objects. The goal Father(john, X) sent to the object john would receive two messages X=bill and X=jill in return.

It would not be possible with this simple implementation to find out who are the parents of bill or jill. This problem can be solved by redundantly nominating more than one argument to serve as the owner of a clause.

Notice that methods can be public or private. Public methods are ones that are known by other objects, which those objects can invoke by sending messages. Private methods are ones that are used only internally.

3.5 Polymorphism

The mapping from input-output logic programs to OO systems illustrates polymorphism. In the context of OO systems, polymorphism is the property that the "same" message can be sent to and be dealt with by objects belonging to different classes; i.e. except for the class of the recipient, everything else about the message is the same:

$$P(o , t_1 , \dots , t_m) \text{ and } P(o', t_1 , \dots , t_m).$$

Thus objects from different classes can respond to the same message using different methods.

Like classes and class hierarchies, polymorphism is neither a unique nor an original feature of OO systems. In the context of logic, polymorphism corresponds to the fact that the same predicate can apply to different sorts of individuals.

3.6 Aspects as Integrity Constraints in Abductive Logic Programming

The mapping from logic programs to OO systems highlights a number of features of logic programming that are not so easily addressed in OO systems. I have already mentioned the problem of appropriately representing relationships, as well as the problem about representing more general logic programs that do not have input-output form. However, a problem that has attracted much attention in software engineering, and which is addressed in ALP, is how to represent *cross-cutting concerns*, which are behaviours that span many parts of a program, but which can not naturally be encapsulated in a single class.

Integrity constraints in ALP have this same character. For example, the concern:

If a person enters a danger zone,
then the person is properly equipped to deal with the danger.

cuts across all parts of a program where there is a sub-goal in which a person needs to enter a danger zone (such as a fireman entering a fire). In ALP, this concern can be expressed as a single integrity constraint, but in normal LP it needs to scattered throughout the program, by adding to any clause that contains a condition of the form:

a person enters a danger zone

an additional condition:

the person is properly equipped to deal with the danger.

In software engineering the problem of dealing with such cross-cutting concerns is the focus of *aspect-oriented programming* (AOP) [5]. AOP seeks to encapsulate such concerns through the introduction of a programming construct called an *aspect.* An aspect alters the behavior of a program by applying additional behavior at a number of similar, but different points in the execution of the program.

Integrity constraints in ALP give a declarative interpretation to aspects. ALP provides the possibility of executing such integrity constraints as part of the process of pre-active thinking [20], to monitor actions before they are chosen for execution. This is like using integrity constraints to monitor updates in a database, except that the updates are candidates to be performed by the program itself.

It is also possible to transform logic programs with integrity constraints into ordinary logic programs without integrity constraints [6,7]. This is similar to the way in which aspects are implemented in AOP. However, whereas in AOP the programmer needs to specify the "join points" where the aspects are to be applied, in logic programming the transformations of [6, 7] can be performed automatically by matching the conditions of integrity constraints with conditions of program clauses.

4 The Relationship between OO systems and ALP Systems

The mapping from input-output logic programs to OO systems can be extended to more general ALP agent systems, and a converse mapping is also possible. These mappings exploit the correspondence between agents and objects, in which both are viewed semantically as individuals, mutually embedded with other individuals in a common, dynamically changing world. Both agents and objects process their interactions with the world, by manipulating sentences in a formal language, encapsulated, and hidden from other individuals.

The biggest difference between logic and objects, and therefore between ALP agents and objects, is their different views of semantic structure. For logic, the world is a relational structure, consisting of individuals and relationships that change over time. Such changes can be modeled by using the possible world semantics of modal logic or by treating situations or events as individuals, but they can also be modeled by using destructive assignment.

With destructive assignment, the world exists only in its current state. Agents perform actions, which initiate new relationships (by adding them) and terminate old relationships (by deleting them), without the world remembering its past. The agents themselves and their relationships with other individuals are a part of this dynamically and destructively changing world.

ALP agents, as well as undergoing destructive changes, can also represent changes internally among their beliefs. Using such syntactic representations of change, they can represent, not only the current state of the world, but also past states and possible future states. We will return to this use of logic to represent change later in the paper.

Whereas logic distinguishes between changes that take place in the semantic structure of the world and changes that are represented syntactically in an agent's beliefs, objects do not. In OO systems, all changes of state are associated with objects. It makes it easy for objects to deal with changes of values of attributes, but more difficult for them to deal with changes of relationships.

The different ways in which logic and objects view the world are reflected in the different ways in which they interact with the world. In OO systems, because the state of the world is distributed among objects as attribute-value pairs, the only way an object can access the current state is by accessing the attribute-value pairs of objects. The only way an object can change the current state is by changing the attribute-value pairs of objects. In some OO languages these operations are carried out by sending and receiving messages. In other OO languages they are performed directly.

ALP agents, on the other hand, interact by observing and acting on the external world. These interactions typically involve observing and changing relationships among arbitrarily many individuals, not only attributes of individual objects. This way of interacting with the world is similar to the way that processes use the Linda tuple-space as a shared environment and to the way that experts use the blackboard in a blackboard expert system.

4.1 Transformation of ALP systems into OO systems

Agents. Our earlier transformation of input-output logic programs into OO systems implicitly treats the owners of clauses as agents. In this transformation, the owner of a clause/belief is selected from among the input arguments of the conclusion of the clause. However, in ALP systems, goals and beliefs are already associated with the agents that are their owners. In transforming ALP systems into OO systems, therefore, it is a simple matter just to treat agents as objects and to treat their goals and beliefs as the objects' methods. In some cases, this transformation of agents into objects coincides with the transformation of input-output logic programs into OO systems. However, in many other cases, it is more general.

The ALP semantic structure. An agent's beliefs include its beliefs about relationships between individuals, expressed as unconditional clauses. The individuals included in these relationships need not explicitly include the agent itself, as in the case of a passenger's belief that there is a fire in a train. These beliefs can be used as methods to respond to requests for information from other agents/objects.

In addition to these syntactic representations of relationships as beliefs, an ALP system as a whole is a semantic structure of relationships between individuals. This semantic structure can also be transformed into objects and their associated attribute-values, similarly to the way in which we earlier associated input-output clauses with owner objects and classes.

However, to obtain the full effect of ALP systems, we need to let each of the individuals o_i in a semantic relationship $P(o_1, \ldots, o_n)$ (expressed as an atomic fact) be an object, and to associate the relationship redundantly with each of the objects o_i as one of its attribute-values. The problem of representing such relationships redundantly has been recognized as one of the problems of OO, and representing relationships as aspects in AOP [9] has been suggested as one way of solving the problem.

Notice that an object corresponding to an agent can contain two representations of the same relationship, one as an attribute-value representation of a semantic relationship, and the other as a very simple method representing a belief. When the two representations give the same result, the belief is true. When they give different results, the belief is false.

Actions. Actions and observations need to be transformed into sending and receiving messages. An action performed by an agent A that initiates relationships

$$P_1(o_{11}, \ldots, o_{1l1})$$
$$\ldots\ldots..$$
$$P_n(o_{n1}, \ldots, o_{nln})$$

and terminates relationships

$$Q_1(p_{11}, \ldots, p_{1k1})$$
$$\ldots\ldots..$$
$$Q_m(p_{m1}, \ldots, p_{mkn})$$

is transformed into a message sent by object A to each object o_{ij} to add $P_i(o_{i1}, \ldots, o_{il1})$ to its attribute-values together with a message sent by A to each object p_{ij} to delete $Q_i(p_{i1}, \ldots, pi_{k1})$ from its attribute-values.

Observations. The transformation of observations into messages is more difficult. Part of the problem has to do with whether observations are active (intentional), as when an agent looks out the window to check the weather, or whether they are passive (unintentional), as when an agent is startled by a loud noise. The other part of the problem is to transform an observation of a relationship between several objects into a message sent by only one of the objects as the messenger of the relationship. To avoid excessive complications, we shall assume that this problem of selecting a single messenger can be done somehow, if necessary by restricting the types of observations that can be dealt with.

An *active observation* by agent A of a relationship $P(o_1, o_2, \ldots, o_n)$ is transformed into a message sent by object A to one of the objects o_j, say o_1 for simplicity, requesting the solution of a goal $P(o_1, o_2', \ldots, o_n')$ and receiving back a message from o_1 with a solution θ, where $P(o_1, o_2, \ldots, o_n) = P(o_1, o_2', \ldots, o_n')\,\theta$.[6]

A *passive observation* of the relationship can be modeled simply by some object o_j sending a message to A of the relationship $P(o_1, o_2, \ldots, o_n)$.

The objects that result from this transformation do not exhibit the benefits of structuring objects into taxonomic hierarchies. This can be remedied by organising ALP agent systems into class hierarchies, similar to sort hierarchies in order-sorted logics. The use of such hierarchies extracts common goals and beliefs of individual

[6] In most OO languages this can be done more directly, without explicitly sending and receiving messages. This more direct access to an object's attribute-values can be regarded as analogous to an agent's observations and actions.

agents and associates them with more general classes of agents. Translating such extended ALP agent systems into OO systems is entirely straight-forward.

4.2 Transformation of OO systems into ALP systems

The semantic structure. Given the current state of an OO system, the corresponding semantic structure of the ALP system is the set of all current object-attribute-values, represented as binary relationships, attribute(object, value), or alternatively as ternary relationships, say as relationship(object, attribute, value).

Agents. We distinguish between *passive objects* that merely store the current values of their attributes and *active objects* that both store their current values and also use methods to interact with the world. Both kinds of objects are treated as individuals in the ALP semantic structure. But active objects are also treated as *agents*.

Messages. The treatment of messages is dealt with case by case:

Case 1. A passive object sends a message to another object. By the definition of passive object, this can only be a message informing the recipient of one of the passive object's attribute-values. The only kind of recipient that can make use of such a message is an active object. So, in this case, the message is an *observation* of the attribute-value by the recipient. The observation is *active* (from the recipient's viewpoint) if the message is a response to a previous message from the recipient requesting the sender's attribute-value. Otherwise it is *passive*.

Case 2. An active object sends a message to another object requesting one of the recipient's attribute-values. This is simply the first half of an *active observation* of that attribute-value. The second half of the observation is a message from the recipient sending a reply. If the recipient does not reply, then the observation fails.

Case 3. An active object sends a message to another object changing one of the recipient's attribute-values. The message is simply an *action* performed on the semantic structure.

Case 4. An active object sends any other kind of message to another active object. The message is a combination of an *action by the sender* and an *observation by the recipient*. The action, like all actions, is performed on the semantic structure. In this case the action adds a ternary relationship between the sender, the recipient and the content of the message to the semantic structure. The observation, like all observations, syntactically records the semantic relationship as a belief in the recipient's internal state. The observation then becomes available for internal processing, using forward and backward reasoning with goals and beliefs to achieve the effect of methods. The recipient may optionally perform an action that deletes the ternary relationship from the semantic structure.

Methods. Finally, we need to implement methods by means of goals and beliefs (or equivalently, for ALP agents, by integrity constraints and logic programs). Recall that, in our analysis, only active objects (or agents) employ methods, and these are used only to respond to messages that are transformed into observations. Other messages, which simply request or change the values of an object's attributes, are transformed into operations on the semantic structure.

We need a sufficiently high-level characterization of such methods, so they can be logically reconstructed. For this reason, we assume that methods can be specified in the following input-output form:

> If observation and (zero or more) conditions,
> then (zero or more) actions.

This is similar to event-condition-action rules in active databases [10] and can be implemented directly as integrity constraints in logical form. However, such specifications can also be implemented at a higher level, by means of more abstract integrity constraints (with more abstract conditions and higher-level conclusions), together with logic programs. At this higher level, an agent implements an active object's method by

- recording the receipt of the message as an observation;
- possibly using the record of the observation to derive additional beliefs;
- possibly using the record of the observation or the derived additional beliefs to trigger an integrity constraint of the form:

  if conditions, then conclusion

- verifying any remaining conditions of the integrity constraint and then,
- reducing the derived conclusion of the constraint, treated as a goal, to sub-goals, including actions.

Beliefs, in the form of logic programs, can be used to reason forwards from the observation and backwards both from any remaining conditions of the integrity constraint and from the conclusion of the integrity constraint. In addition to any actions the agent might need to perform as part of the specification of the method, the agent might also send requests to other agents for help in solving sub-goals, in the manner of the object-oriented logic programs of section 3.

All messages that do not request or change attribute-values are treated by the recipient uniformly as observations. If the message is a request to solve a goal, then the recipient records the request as an observation and then determines whether or not to try to solve the goal, using an integrity constraint such as:

> If an agent asks me to solve a goal,
> and I am able and willing to solve the goal for the agent,
> then I try to solve the goal and I inform the agent of the result.

The recipient might use other integrity constraints to deal with the case that the recipient is unable or unwilling to solve the goal.

Similarly, if the message is a communication of information, then the recipient records the communication as an observation and then determines whether or not to add the information to its beliefs. For example:

> If an agent gives me information,

and I trust the agent,
and the information is consistent with my beliefs,
then I add the information to my beliefs.

The input-output specification of OO methods that we have assumed is quite general and hopefully covers most sensible kinds of methods. As we have seen, the specification has a direct implementation in terms of abductive logic programming. However, as we have also noted earlier, other implementations in other computer languages are also possible, as long as they respect the logical specification.

Classes. Methods associated with classes of objects and inherited by their instances can similarly be associated with sorts of ALP agents. This can be done in any one of the various ways mentioned earlier in the paper. As remarked then, this requires an extension of ALP agents, so that goals and beliefs can be associated with sorts of agents and acquired by individual agents. There is an interesting research issue here: whether sentences about several sorts of individuals can be represented only once, or whether they need to be associated, possibly redundantly, with owner classes/sorts.

5 Local versus Global Change

One of the attractions of object-orientation is that it views change in local, rather than global terms. In OO the state of the world is distributed among objects as the current values of their attributes. Change of state is localized to objects and can take place in different objects both concurrently and independently.

Traditional logic, in contrast, typically views change in global terms, as in the possible-worlds semantics of modal logic and the situation calculus. Modal logic, for example, deals with change by extending the static semantics of classical model theory to include multiple (possible) worlds related by a temporal, next-state accessibility relation. Semantically, a change of state due to one or more concurrent actions or events, is viewed as transforming one global possible world into another global possible world. Syntactically, change is represented by using modal operators, including operators that deal with actions and events as parameters, as in dynamic modal logic.

The situation calculus [11] similarly views change as transforming one global possible world (or situation) into another. Semantically, it does so by reifying situations, turning situations into individuals and turning the next-state accessibility relation into a normal relation between individuals. Syntactically, it represents change by using variable-free terms to name concrete situations and function symbols to transform situations into successor situations.

It was, in part, dissatisfaction with the global nature of the possible-worlds semantics that led Barwise and Perry to develop the situation semantics [12]. Their situations (which are different from situations in the situation calculus) are semantic structures, like possible-world semantic structures, but are partial, rather than global.

It was a similar dissatisfaction with the global nature of the situation calculus that led us to develop the event calculus [13]. Like situations in the situation calculus, events, including actions are reified and represented syntactically. The effect of actions/events on the state of relationships is represented by an *axiom of persistence* in logic programming form:

```
A relationship holds at a time $T_2$
if an event happens at a time $T_1$ before $T_2$
and the event initiates the relationship
and there is no other event
    that happens at a time after $T_1$ and before $T_2$ and
    that terminates the relationship.
```

Like change of state in OO, change in the event calculus is also localised, but to relationships rather than to objects. Also as in OO, changes of state can take place concurrently and independently in different and unrelated parts of the world. Each agent can use its own local clock, time-stamping observations as they occur and determining when to perform actions, by comparing the time that actions need to be performed with the current time on its local clock.

The event calculus is a syntactic representation, which an agent can use to reason about change. It can be used to represent, not only current relationships, but also past and future relationships, both explicitly by atomic facts and implicitly as a consequence of the axiom of persistence. However, the event calculus does not force an agent to derive current relationships using the persistence axiom, if the agent can observe those relationships directly, more efficiently and more reliably instead.

The event calculus is not a semantics of change. However, in theory, if events are reified, then the use of the event calculus to reason about change should commit an agent to a semantic structure in which events are individuals. But this is the case only if all symbols in an agent's goals and beliefs need to be interpreted directly in the semantic structure in which the agent is embedded. If some symbols can be regarded as defined symbols, for example, then they need not be so interpreted. Alternatively, in the same way that in physics it is possible to hypothesize and reason with the aid of theoretical particles, which can not be observed directly, it may also be possible in the event calculus to represent and reason about events without their being observable and without their corresponding to individuals in the world of experience.

In any case, the event calculus is compatible with a semantic structure in which changes in relationships are performed destructively, by deleting (terminating) old relationships and adding (initiating) new relationships. These destructive changes in the semantic structure are the ones that actually take place in the world, as opposed to the representation of events and the derivations of their consequences using the axiom of persistence, which might take place only in the mind of the agent.

6 Related Work

There is a vast literature dealing with the problem of reconciling and combining logic programming and object-orientation, most of which was published in the 1980s, when the two paradigms were still contending to occupy the central role in Computing that OO occupies today. Most of this early literature is summarized in McCabe's [14].

Perhaps the most prominent approach among the early attempts to reconcile logic programming and objects was the concurrent object-oriented logic programming approach exemplified by [15]. In this approach, an object is implemented as a process that calls itself recursively and communicates with other objects by instantiating shared variables. Objects can have internal state in the form of unshared arguments that are overwritten in recursive calls. Although this approach was inspired by logic programming it ran into a number of semantic problems, mainly associated with the use of committed choice. The problem of committed choice is avoided in ALP systems by incorporating it in the decision making component of individual agents.

In McCabe's language, *L&O* [14], a program consists of a labelled collection of logic programs. Each labelled logic program is like a set of beliefs belonging to the object or agent that is the label. However, in *L&O,* objects/agents interact by sending messages to other agents, asking for their help in solving sub-goals. This is like the object-oriented logic programs of section 3.4. It is also similar to the way multi-agent systems are simulated in GALATEA [22].

ALP systems differ from *L&O*, therefore, primarily in their use of a shared environment instead of messages. This use of a shared environment is similar to the use of tuple-spaces in Linda [2]. In this respect, therefore, ALP systems are closest to the various systems [16-18] that use a Linda-like environment to coordinate parallel execution of multiple logic programs. ALP systems can be viewed, therefore, as providing a logical framework in which the shared environment in such systems can be understood as a dynamic semantic structure.

A different solution to the problem of reconciling logic and objects is the language LO [19], which is a declarative logic programming language using linear logic. The language is faithful to the semantics of linear logic, which however is quite different from the model-theoretic semantics of traditional logic. Communication between objects in LO is similar to that in Linda.

7 Conclusions

In this paper, I have explored some of the relationships between OO systems and ALP systems, and have argued that ALP systems can combine the semantic and syntactic features of logic with the syntactic structuring and dynamic, local behaviour of objects. I have investigated a number of transformations, which show how OO systems and ALP systems can be transformed into one another. These transformations are relatively straight-forward, and they suggest ways in which the two kinds of system are related. Among other applications, the transformations can be used to embed one kind of system into the other, for example along the lines of [22], and therefore to gain the benefits of both kinds of systems.

However, the transformations also highlight a number of important differences, including problems with the treatment of relations and multi-owner methods in OO systems in particular. On the other hand, they also identify a number of issues that need further attention in ALP systems, including the need to clarify the distinction between active and passive observations, to organise agents into more general agent hierarchies, and possibly to structure the shared semantic environment, to take account of the fact that different agents can more easily access some parts of the environment than other parts. In addition, it would be useful to make the relationships between OO systems and ALP systems explored in this paper more precise and to prove them more formally.

As a by-product of exploring the relationships between logic and objects, the transformations also suggest a relationship between logic and Linda. On the one hand, they suggest that Linda systems can be understood in logical terms, in which tuple-spaces are viewed as semantic structures and processes are viewed as agents interacting in this shared semantic environment. On the other hand, they also suggest that ALP systems can be generalised into Linda-like systems in which different processes can be implemented in different languages, provided that the external, logical specification of the processes is unaffected by their implementation.

Acknowledgements I am grateful to Ken Satoh at NII in Tokyo, for valuable discussions and for providing a congenial environment to carry out much of the work on this paper. I would also like to thank Jim Cunningham, Jacinto Davila and Maarten van Emden for valuable comments on an earlier draft of this paper.

It is my pleasure to dedicate this paper to Gigina Aiello, for her inspiration in the early stages of this work, when we first started thinking about logic-based multi-agent systems in the Compulog Project in the late 1980s and early 1990s.

References

1. Kowalski, R., Sadri, F.: From Logic Programming towards Multi-agent Systems. Annals of Mathematics and Artificial Intelligence. (25) (1999) 391- 419
2 Gelernter, D.: Generative Communication in Linda, ACM Transactions on Programming Languages, (7)1 (1985) 80-112
3. Brown, G. Yule, G.: Discourse Analysis. Cambridge University Press (1983)
4. Williams, J.: Style: Towards Clarity and Grace. Chicago University Press (1990)
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming, vol.1241, (1997) 220–242
6. Kowalski, R., Sadri, F.: Logic Programming with Exceptions. New Generation Computing, (9)3,4 (1991) 387-400
7. Toni, F., Kowalski, R.: Reduction of Abductive Logic Programs to Normal Logic Programs. Proceedings International Conference on Logic Programming, MIT Press (1995) 367-381
8. Engelmore, R.S., Morgan, A. (eds.): Blackboard Systems. Addison-Wesley (1988)
9. Pearce, D.J., Noble, J.: Relationship Aspects. Proceedings of the ACM conference on Aspect-Oriented Software Development (AOSD'06). (2006) 75-86
10. Paton, N.W., Diaz, O.: Active Database Systems. ACM Computing Surveys 31(1) (1999)
11. McCarthy, J., Hayes, P.: Some Philosophical Problems from the Standpoint of AI. Machine Intelligence. Edinburgh University Press (1969)
12. Barwise, J., Perry, J.: Situations and Attitudes. MIT-Bradford Press, Cambridge (1983)
13. Kowalski, R., Sergot, M.: A Logic-based Calculus of Events. New Generation Computing. (4)1 (1986) 67-95
14. McCabe, F.G.: Logic and Objects. Prentice-Hall, Inc. Upper Saddle River, NJ (1992)
15. Shapiro, E., Takeuchi, A.: Object-Oriented Programming in Concurrent Prolog, New Generation Computing 1(2) (1983) 5-48

16. Andreoli, J.M., Hankin, ., Le Métayer, D. (eds.): Coordination Programming: Mechanisms, Models and Semantics - Imperial College Press, London (1996)
17. Brogi, A., Ciancarini, P.: The Concurrent Language, Shared Prolog. ACM Transactions on Programming Languages and Systems, 13(1) (1991) 99–123
18. De Bosschere, K., Tarau, P.: Blackboard-Based Extensions in Prolog. Software - Practice and Experience 26(1) (1996) 49-69
19. Andreoli, J.-M., Pareschi, R.: Linear Objects: Logical Processes with Built-In Inheritance. New Generation Computing, 9 (1991) 445-473
20. Kowalski, R.: How to be Artificially Intelligent. In Toni, F., Torroni, P. (eds.): Computational Logic in Multi-Agent Systems. LNAI 3900, Springer-Verlag (2006) 1-22
21. Armstrong, D.J.: The Quarks of Object-Oriented Development. Communications of the ACM 49 (2) (2006) 123-128
22. Davila,J., Gomez, E., Laffaille, K., Tucci, K., Uzcategui, M.: Multi-Agent Distributed Simulations with GALATEA. In Boukerche, A., Turner, T., Roberts, D., Theodoropoulos, G. (eds): IEEE Proceedings of Distributed Simulation and Real-Time Applications. IEEE Computer Society (2005) 165-170