Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents

Robert Kowalski and Fariba Sadri
Department of Computing, Imperial College London, 180 Queens Gate, London SW7 2AZ
{rak, fs}@doc.ic.ac.uk

Abstract. In this paper we argue the case for integrating the distinctive functionalities of logic programs and production systems within an abductive logic programming agent framework. In this framework, logic programs function as an agent's beliefs and production rules function as the agent's goals. The semantics and proof procedures are based on abductive logic programming, in which logic programs are integrated with integrity constraints that behave like production rules.

Similarly to production systems, the proof procedure is an operational semantics, which manipulates the current state of a database, which is modified by actions implemented by destructive assignment. The semantics can be viewed as generating a model, based on the sequence of database states and logic program, which makes the production rules true.

Keywords: Abductive logic programming, Production systems, Integrity constraints, agents.

1 Introduction

Rules are the basic form of knowledge representation in many areas of Artificial Intelligence, including both production systems and logic programming, and more recently in BDI (Belief Desire Intentions) agent languages. Despite their wide-spread use, there is a great deal of confusion between the different kinds of rules, and little agreement about the relationship between them.

In this paper we argue that production rules and logic programming rules have complementary characters and that one cannot usefully be reduced to the other. We show how abductive logic programming (ALP) combines the two kinds of rules in a single unified framework. The ALP framework gives a model-theoretic semantics to both kinds of rules and provides them with powerful proof procedures, combining both backward and forward reasoning. We present evidence from BDI agents, deductive databases and psychological experiments to support the distinct nature of the two kinds of rules.

We discuss the impact of including the two kinds of rules in a production system or agent cycle, which embeds the rules in a destructively changing environment, which is like a production system working memory. In this embedding, the environment can be viewed as the semantic structure that gives meaning to the two kinds of rules. Although the rules themselves need to be understood as explicitly or implicitly representing change of state, the fact that the environment changes

destructively and exists at any given time only in its current state gives an efficient solution to the frame problem.

We assume the reader is familiar with the basic concepts of logic programming and production systems, although not necessarily with all of their technicalities.

1.1 Confusions

The most popular textbook on Artificial Intelligence Russell and Norvig [45] views production rules as just logical conditionals used to reason forward (page 286). In contrast, in one of the main textbooks on Cognitive Science, Thagard [50] argues that "Rules are if-then structures …very similar to the conditionals…, but they have different representational and computational properties." (page 43). "Unlike logic, rule-based systems can also easily represent strategic information about what to do. Rules often contain actions that represent goals, such as *IF you want to go home for the weekend and you have bus fare, THEN you can catch a bus*." (page 45).

Thagard [50] characterizes Prolog as "a programming language that uses logic representations and deductive techniques". Simon [48], on the other hand, includes Prolog "among the production systems widely used in cognitive simulation."

There is a similar confusion in the field of agents. Rao [41], for example, characterises AgentSpeak as similar to logic programming. But in his comparison, he considers only the similarities between the operational semantics of plans in AgentSpeak and the execution of clauses in logic programming. He ignores the declarative semantics of logic programs (LPs).

1.2 Production Systems and Logic Programs in Practice

There have been many theoretical studies of the relationship between production rules and logic programs, which we discuss below in Section 2. Most of this work has been focussed on giving a declarative semantics to production systems by translating them into logic programs. However, there seems to have been little attention paid to the way in which logic programs and production rules are used in practice, and consequently little attempt to use this practice to guide the theoretical analysis. We argue that in practice, the two kinds of rules have both distinct and overlapping functionalities, and that the distinct functionalities are lost by translating one kind of rule into the other. We will show that abductive logic programming (ALP) both capitalises on the distinct functionalities and eliminates the overlap.

We argue that, in addition to the production rule (PR) cycle and destructively changing database, which are absent in LP, PRs offer three distinct functionalities: reactive rules that implement stimulus-response associations; forward chaining logic rules; and goal-reduction rules.

Reactive rules are, arguably, the most distinctive type of production rules, which are responsible for their general characterisation as *condition-action* rules. This kind of rule typically has implicit or *emergent* goals. For example, the rule *if a car coming towards you then get out of its way* has the implicit goal *to stay safe*. Reactive rules provide a functionality that is not directly available in logic programming.

The second kind of rule, for example *if X is a cat then X is an animal* uses forward chaining to implement forward reasoning with a logical conditional. It is probably this kind of rule that gives the impression that production rules are just conditionals used to reason forward.

It is the third kind of rule, exemplified by Thagard's example of the goal-reduction rule "*IF you want to go home for the weekend and you have bus fare, THEN you can catch a bus*.", that overlaps the most with logic programming. In logic programming, such strategic rules would be obtained by reasoning backward with the clause *you go home for the weekend if you have bus fare and you catch a bus*. The two best known cognitive models of human thinking, SOAR [34] and ACT-R [3], are based on production systems and focus on the use of production rules for goal-reduction.

Logic programming has its own confusions, mostly about whether clauses are to be understood declaratively or procedurally. The purely declarative interpretation of clauses, which is neutral about reasoning method, is probably the one that is most attractive to its admirers. It is well-suited for high-level program specifications and for certain applications where efficiency is not a major concern.

However, it is probably the procedural interpretation, in which clauses are used to obtain goal-reduction by backward reasoning, that is the main way in which logic programs are used in practice. This is also where there is the greatest overlap with production rules. Arguably, logic programs with backward reasoning are more suitable for this purpose than production rules with forward reasoning, because logic programs can also be interpreted declaratively. The declarative interpretation of logic programs makes it possible to give goal-reduction procedures the declarative semantics that is missing with production rules.

The confusion between the declarative and procedural uses of logic programs and how best to combine them is well-known even though it is not very well solved. However, there is another use of logic programs that has received less attention, and is perhaps even more confusing. It is the use of logic programs for forward reasoning. This use is not very common in practice, but is prevalent in theoretical investigations of logic programming. We will see that in ALP, clauses can be used to reason both backward and forward.

1.3 Combining Production Systems and Logic Programs

Broadly speaking, there are four motivations for combining PRs and LPs:

1. To eliminate the overlap between forward logic rules in PRs and forward/declarative clauses in LP. For example, one very simple combination of PRs and LPs is to use LPs to define ramifications of the working memory/database. Then existing PRs could simply query a deductive database rather than a relational database. This would hand over the forward reasoning logic rules from the PR to the LP component. Moreover, it would allow the decision about how to execute the ramifications to be taken by the implementation. The declarative semantics of ramifications would be compatible with executing them forward, backward, or any combination of the two.

2. To eliminate the overlap for goal-reduction, by using LPs for this purpose. Using LP for goal-reduction provides system support for managing goals as and-or trees, which is missing in production systems. Whereas most production systems just treat goals as ordinary facts in the working memory, SOAR and ACT-R manipulate them in goal stacks. Using LP for goal-reduction allows the declarative nature of LP clauses to be exploited, so they can be used either for goal-reduction or for forward reasoning, as the context requires.

3. To provide a declarative semantics for PRs and for the combination of LP and PRs. Declarative semantics provides an independent specification for implementations, as illustrated by the discussion above about the implementation of ramifications. Declarative semantics also clarifies the nature of PRs as a representation language. Without a declarative semantics, which establishes a relationship between syntactic expressions and semantic structures, of the kind provided by the model theory of logic, the term “representation” has no meaning.

Our proposal is to combine LP and PR in the same way that ALP combines LP and integrity constraints (ICs), and to use the model-theoretic semantics of ALP to give a model-theoretic semantics to the combination of LP and PR. We will show that integrity constraints in ALP, especially when embedded in ALP agents, generalise production rules to include *condition-goal* rules, where the *goal* is like the body of a plan in BDI agents.

4. To provide a cycle and destructive database of facts, missing in LP. Without a cycle, LP is both closed and passive – closed because logic programs cannot be updated by the environment, and passive because they cannot perform updates on the environment. Without a destructive database of facts, LP suffers from the inefficiencies of the frame problem.

2 Other approaches

Typically PRs, as well as event-condition-action (ECA) rules and active integrity constraints are defined by means of an operational semantics based on state transitions. However several authors have studied the relationship between these various kinds of rules and LP, with the aim of providing the rules with a declarative LP-based semantics. In the majority of these approaches PRs, ECA rules or active integrity constraints are mapped into LP to provide them with LP-based semantics. To our knowledge, there has been no proposal that would accommodate both LP and PR (or ECA rules or active integrity constraints) side-by-side with an integrated semantics or proof procedure that would exploit the strengths of both paradigms.

Raschid [52] combines LPs and ICs, but focuses on only two functionalities of PRs, namely on their use as reactive rules and as forward logic rules. She represents rules that add facts as LPs, and rules that delete facts as ICs. She then transforms their combination into LP, and uses the fixed point semantics of LP to chain forward and thereby simulate the production system cycle.

Ceri and Widom [7] and Ceri *et al.* [6] implement ICs by PRs. Dung and Mancarella [14] use an argumentation theoretic framework to provide semantics for PRs with negation as failure.

Caroprese *et al.* [5] transform active integrity constraints into LPs. They characterise the set of "founded" repairs for the database as the stable model of the database augmented by the LP representation of the active integrity constraints. Fraternali and Tanca [17] also consider active databases but provide a logic-based *core* syntax for representing low-level, procedural features of active database rules. They provide procedural semantics for core rules and show how this can capture the procedural semantics of known active database systems.

Most other work regards the cycle and actions in condition-action rules and ECA rules declaratively as performing a change of state. Zaniolo [54], for example, uses a situation calculus-like representation with frame axioms, and reduces PRs and ECA rules to LPs. Statelog [53] also uses a situation-calculus-like representation for the succession of database states. Like Zaniolo, Statelog represents PRs and ECAs as LPs, and gives them LP-based semantics. Neither is concerned with the role of ICs or with the use of LPs and PRs for goal-reduction.

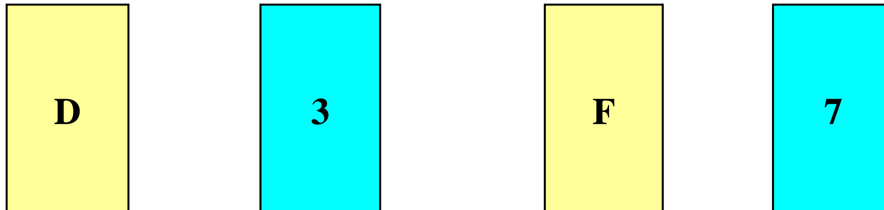
Fernandes *et al.* [16] also view ECAs in terms of change of state, but use the event calculus as the basis for an ECA language coupled with a deductive database. The event calculus is used to evaluate the condition part of the ECA rules and to provide a specification for the effects of executing the action part. The ECA language also allows the recognition of complex events from an event history.

ERA (Evolving Reactive Algebraic Programs), developed by Alferes *et al.* [1], extends the dynamic logic programming system EVOLP [2] by adding complex events and actions as well as external actions. ERA combines ECA and LP rules, and the firing of the ECA rules can generate actions that add or delete ECA or LP rules, as well as external actions. In the operational semantics the ECA and LP rules maintain their distinct characteristics, but in the declarative semantics the ECA rules are translated to LP. The declarative semantics is based on a variant of stable models developed for EVOLP.

3 The selection task

Psychological evidence from the selection task suggests that people reason differently with two kinds of conditionals. One school of thought is that the difference depends, at least in part, on whether conditionals are interpreted descriptively or deontically. We will argue that descriptive conditionals are like logic programs, and deontic conditionals are like integrity constraints in abductive logic programming.

In Wason's original selection task, there are four cards, with letters on one side and numbers on the other. The cards are lying on a table with only one side of each card showing:



The task is to select those and only those cards that need to be turned over, to determine whether the following conditional is true:

If there is a D on one side,
then there is a 3 on the other side.

Variations of this experiment have been performed numerous times. The surprising result is that only about 10% of the subjects give the correct answer according to the norms of classical logic.

Almost everyone recognizes, correctly, that the card showing D needs to be turned over, to make sure there is a 3 on the other side. Most people also recognize, correctly, that the card showing F does not need to be turned over. But many subjects also think, incorrectly, that it is necessary to turn over the card showing 3, to make sure there is a D on the other side. This is logically incorrect, because the implication does not claim that *conversely*:

If there is a 3 on one side,
then there is a D on the other side.

Only a few subjects realise that it is necessary to turn over the card showing 7, to make sure that D is not on the other side. It is necessary to turn over the 7, because the original implication is logically equivalent to its *contrapositive*:

If the number on one side is not 3 (e.g. 7),
then the letter on the other side is not D.

It has been shown that people perform far better, according to the norms of classical logic, when the selection task experiment is conducted with certain other formulations of the problem that are formally equivalent to the card version of the task. The classic experiment of this kind considers the situation in which people are drinking in a bar, and the subject is asked to check whether the following conditional holds:

If a person is drinking alcohol in a bar,
then the person is at least eighteen years old.

Again the subject is presented with four cases to consider, but this time instead of four cards there are four people. We can see what two people are drinking, but cannot see how old they are; and we can see how old two people are, but not what they are

drinking. In contrast with the card version of the selection task, most people solve the bar version correctly. They realise that it is necessary to check the person drinking alcohol to make sure that he is at least eighteen years old, and to check the person under eighteen to make sure that she is not drinking alcohol. They also realise that it is not necessary to check the person who is eighteen years old or older, nor the person who is drinking a non-alcoholic beverage.

Cognitive psychologists have proposed a bewildering number of theories to explain why people are so much better at solving such versions of the selection task compared with the original card version. One of the most influential of these is the theory put forward by Cheng and Holyoak [8] that people tend to reason in accordance with classical logic when conditionals involve *deontic* notions concerned with permission, obligation and prohibition. However, except for [49] and [25], there has been little attempt to explain why people reason as they do with descriptive variants of the selection task, such as the card version.

Stenning and van Lambalgen [49] propose that understanding and solving the selection task is a two stage process: interpreting the conditional and then reasoning with the interpretation. They argue that people interpret conditionals of the kind involved in the card version of the task as logic programs. Interpreted as a logic programming clause, a conditional is understood, according to the *completion semantics*, as the if-half of a definition in if-and-only-if form [9]. The completion semantics entails the converse of conditionals in the selection task and inhibits the application of reasoning with contrapositives. This is exactly the kind of reasoning most people display in the card version of the selection task.

Stenning and vanLambalgen also argue that it is natural to interpret conditionals of the kind involved in the bar version of the task in deontic logic. However, Kowalski [25] has argued that the deontic interpretation can be obtained more simply by interpreting conditionals as integrity constraints.

It is curious that, although production systems, such as SOAR and ACT-R, are widely used in cognitive psychology as a model of human thinking, it seems that conditionals in the form of condition-action rules have not been studied in relation to the selection task.

4 Intelligent agents

The psychological evidence that people reason differently with descriptive and deontic conditionals is mirrored by the notion that the mental state of an intelligent agent is best understood as having separate goals and beliefs. An agent's beliefs represent the way things are, and its goals represent the way the agent would like them to be. Thus beliefs have a descriptive character, whereas goals have a prescriptive or deontic character. The BDI (Belief, Desire, Intention) [4] model of agents adds to beliefs and desires (or goals) the notion of intention, which is an agent's plan of actions for achieving its goals. Intentions are derived from goals using beliefs to reduce goals to subgoals.

Arguably, the most influential of the BDI agent models is that of Rao and Georgeff [42] and its successors dMARS [13] and AgentSpeak [41]. The abstract agent

intermediate language AIL of Dennis et al. [12] is an abstraction of these languages, based mainly on AgentSpeak and its successors.

The earliest BDI agent systems were specified in multi-modal logics, with separate modal operators for goals, beliefs and intentions. However, their procedural implementations bore little resemblance to their logical specifications. AgentSpeak abandoned the attempt to relate the modal logic specifications with their procedural implementations, observing instead that "…one can view agent programs as multi-threaded interruptible logic programming clauses". This abandonment of modal logic specifications is inherited by AgentSpeak's successors and their abstraction AIL.

However, this view of AgentSpeak in logic programming terms applies only to the procedural interpretation of clauses. In fact, programs in AgentSpeak are better viewed as a generalisation of production rules than as variant of logic programming. AgentSpeak programs, also called plans, have the form:

Event E: conditions C $\Leftarrow$ *goals G and actions A.*

AgentSpeak plans manipulate a "declarative" database, like the working memory in production systems. The database contains both *belief literals* (atoms and negations of atoms) and *goal atoms*. The belief literals represent the current state of the environment and are added and deleted destructively, simulating the execution of atomic actions. Goal atoms are added when they are generated as sub-goals, and deleted when they are solved.

The event *E* in the head of a plan can be the addition or deletion of a belief or of a goal. Plans are embedded in a cycle similar to the production system cycle, and are executed in the direction in which they are written. With the arrow written backwards, the execution of plans can be viewed as backward chaining. However, if the arrow is reversed, their execution can be viewed as forward chaining. No matter how their execution is viewed, plans have only an operational semantics.

The following are examples of possible AgentSpeak plans:

+ there is a fire: true $\Leftarrow$ *+there is an emergency.*
+?there is an emergency: true $\Leftarrow$ *? there is a fire.*
+! there is an emergency: true $\Leftarrow$ *! there is a fire.*

Observations and actions do not have associated times, and the database provides only a snapshot of the current state of the world. To compensate for this lack of a temporal representation, the prefixes +,-, *!,* and *?* are used to stand for *add, delete, achieve,* and *test*, respectively.

Notice that the first plan behaves like a logical conditional used to reason forwards, but in the opposite direction of the arrow, to conclude there is an emergency if it is observed that there is a fire. The other two plans are goal-reduction rules, one for testing whether there is an emergency and the other for creating an emergency.

In general, in the case where a plan has the form:

Goal E: conditions C $\Leftarrow$ *goals G and actions A*

and the triggering event *E* is the addition of a goal, the plan can be reformulated as a logic programming clause:

E' if C' and G' and A'and temporal constraints

where the prefixed predicates of AgentSpeak are replaced by predicates with explicit associated times. The corresponding clause subsumes the behaviour of the plan, but also has a declarative reading.

In the simple example of the three plans above, the corresponding clause is:

there is an emergency at time T if there is a fire at time T.

Represented in this way, the clause can be viewed as defining a ramification, which views fires more abstractly as emergencies.

Thus, although BDI agent models were inspired by the modal logic representation of goals and beliefs, this inspiration has largely been lost in recent years. Most agent systems today represent goals as facts, mixed with belief facts in database or represented in a separate stack, as in ACT-R and SOAR. Belief facts and goal facts are manipulated uniformly by procedures, often called plans, which generalise production rules.

The only kind of goal that can easily be represented as a fact in a database or in a goal stack, in this way, is an *achievement goal*, which is a one-off problem to be solved, including the problem of achieving some desired future state of the world. The higher-level notion of *maintenance goal*, which persists over all states of the world, is lost in the process.

In ALP agents, as we will see below, maintenance goals are integrity constraints, which have the form of universally quantified conditionals with existentially quantified conclusions. Thus maintenance goals are higher-level than achievement goals in ALP, because an achievement goal is derived as an instance of the conclusion of a maintenance goal, whenever an instance of the conditions of the maintenance goal are satisfied. For example, given the maintenance goal:

For all times T_1
If there is an emergency at time T_1 *then there exists a time* T_2 *such that*
I get help at time T_2 *and* $T_1 < T_2$

and an emergency at some specific time t_1, forward reasoning in ALP would derive the achievement goal of getting help at some later time. The later time could be bounded by an additional conjunct in the conclusion, or it could be left to the decision-making component of the agent cycle to take into account how urgently the achievement goal needs to be accomplished.

In AgentSpeak and its successors, maintenance goals and goal-reduction rules are just different kinds of plans. Our aim is to restore the high level distinction between goals and beliefs, to recognise the importance of maintenance goals in particular, to combine the distinctive forms of reasoning appropriate to the distinction between goals and beliefs, and to give their combination a logical, model-theoretic semantics. For this purpose we interpret beliefs as logic programs, goals as integrity constraints,

and combine goals and beliefs in the way that logic programs and integrity constraints are combined in abductive logic programming.

5 Deductive databases

The combination of logic programs and integrity constraints in ALP evolved from their relationship in deductive databases. The semantics of integrity constraints and the development of proof procedures for constraint satisfaction were active research areas in deductive databases in the 1980s.

The distinction between a database and its integrity constraints is intuitively clear in database systems, where integrity constraints have the same semantics as database queries. But, whereas *ad hoc* queries are concerned with properties that hold in a given state of the database, integrity constraints are persistent queries that are intended to hold in all states of the database. *Ad hoc* queries can be viewed as achievement goals, and integrity constraints can be viewed as maintenance goals and include prohibitions as a special case. The database itself can be thought of as a set of beliefs. Thus conventional database systems can be viewed as passive agents, which are open to updates from the environment, but are unable to perform actions themselves.

In relational databases, there is a clear distinction between the syntax of beliefs in the database and the syntax of goals. Beliefs are simple, ground, atomic sentences. Goals, both ad hoc queries and persistent integrity constraints, are sentences of first-order logic. However, the syntactic distinction is less clear in deductive databases, where the database consists of both ground facts and more general logic programs (also called deduction rules). The distinction is complicated by the fact that it is often natural to express both deduction rules and integrity constraints in a similar conditional form. Informal criteria for distinguishing between deduction rules and integrity constraints were proposed by Nicolas and Gallaire [37]. Consider, for example, the two conditionals:

The bus leaves at time X:00 if X is an integer and $9 \leq X \leq 18$.

If the bus leaves at time X:00, then for some integer Y,
the bus arrives at its destination at time X:Y and $20 \leq Y \leq 30$

The first conditional defines bus departure times constructively and therefore can function as a general rule in a deductive database. However, the second conditional has an existential quantifier in the conclusion, which means that it cannot be used to define data, but can only be used to constrain data, as an integrity constraint. In a passive database, the integrity constraint can be used to check updates to the database. But in an agent, the integrity constraint can be used as a maintenance goal, to generate achievement goals, which can then be reduced to plans of action. Thus an agent can be thought of as an active database, and its maintenance goals can be regarded as active database rules.

Several competing views of the semantics of integrity constraints were intensively investigated in the 1980s. The two main views, to begin with, were the consistency view and the theorem-hood view, both of which were defined relative to the completion of the database. In the *consistency view*, an integrity constraint is satisfied

if it is consistent with the completion of the database. In the *theorem-hood view*, it is satisfied if it is a theorem, logically entailed by the completion.

Reiter [44] proposed an *epistemic view* of integrity constraints, according to which integrity constraints are statements about what the database knows. For example, the integrity constraint:

If X is an employee then for some integer Y
X has social security number Y

would be interpreted as:

If the database knows that X is an employee then for some integer Y
the database knows that X has social security number Y

However, Reiter [44] also showed that all three views are equivalent in many cases for databases augmented with the *closed world assumption* [43] which is the set of all the negations of atomic sentences that are not entailed by the database. For relational databases, the three views are also equivalent to the standard view in relational databases that a database satisfies an integrity constraint if it is *true* in the database regarded as a Herband model.

More generally, the four views of integrity satisfaction (consistency, theorem-hood, epistemic and truth-theoretic) coincide for any database whose closure has a single model, In the case of Horn clause databases, the four views are equivalent to the view that an integrity constraint is satisfied if (and only if) it is true in the unique minimal model of the set of Horn clauses.

However, whether or not the different views of integrity satisfaction are equivalent for a given database, it is generally accepted that queries and integrity constraints have the same semantics. Therefore, the most obvious way to check integrity satisfaction is to treat each integrity constraint as a query, using the same procedure for integrity checking as for query evaluation. The problem with this approach, is that in a dynamic setting, where the current database state is largely identical to the previous state, much of the work involved in processing the constraints in a new state duplicates the work performed in the previous state.

To alleviate this problem, the vast majority of integrity checking procedures developed in the 1980s incrementally checked the integrity of a new state of the database, assuming that the integrity constraints already hold in the previous state. As a consequence any new violation of integrity must be due to the update itself, and integrity checking can be focused on the update and its consequences. This focus can be achieved by using resolution to perform forward reasoning.

The combination of a sequence of updates to a database and forward reasoning triggered by updates can be viewed in terms that resemble the production system cycle. Early work on such an approach for integrity checking in deductive databases includes the integrity checking procedure of Sadri and Kowalski [46, 47, 30]. In this approach integrity constraints are formalised as denials (clauses with conclusion *false*), for example:

If X is an employee and not X has a social security number then false

Transactions are sets of literals, in which the positive atoms represent requested additions, and the negated atoms represent requested deletions. In the simple case where all the requested updates are additions:

> If an update matches one of the conditions of a clause or integrity constraint,
> forward reasoning (via resolution) is performed to generate the resolvent.
> SLDNF is used to try to verify the remaining conditions of the resolvent.
> If the conditions are verified, then
> the instantiated conclusion is added as a new update.
> If the new update is *false*, the procedure terminates,
> and integrity has been violated.
> Otherwise, the procedure is repeated, and
> the new update is treated as an (implicit) update in the same transaction.
> If the procedure terminates without generating false
> then the transaction satisfies the integrity constraints.

If any of the updates is a deletion, a similar procedure is applied with an extended resolution step that allows resolution with negative literals. Here is an example without deletions:

ICs:	*If P and not R then false*	*If S and Q then false*	
Database:	*S if M*	*Q if T*	*R if T*
Updates:	{*P, T*}		

Forward reasoning from the two updates produces two resolvents:

> *if not R then false*
> *if S then false.*

Forward reasoning from the update *P* produces the first resolvent, using the first IC. Forward reasoning from the update *T* produces the second resolvent, using the second database clause to derive *Q* and then using the second IC.

The SLDNF evaluation of the two conditions *not R* and *S* terminates unsuccessfully, and therefore the integrity checking procedure terminates successfully.

In the Sadri Kowalski (SK) integrity checking procedure sketched above, ICs are represented as denials, which conceptually represent prohibitions. However, the procedure can be modified easily to deal with ICs in conditional form. Integrity checking with ICs in conditional form highlights their relationship with production rules, and was introduced in the IFF proof procedure for ALP.

6 Abductive Logic Programming

Deductive databases and the closely related Datalog were sidelined in the late 1980s with the arrival of object-oriented and active databases. As we have seen in section 2, some of the attention of the Datalog community turned to the problem of providing a

declarative semantics for active database rules. In the meanwhile, some of the work on integrity checking in deductive databases contributed to the development of abductive logic programming (ALP) [19].

ALP can be viewed as a variant of deductive databases in which integrity constraints are used actively to generate new facts that are candidates for addition to the database. However, ALP is normally viewed as an extension of logic programming, combining *closed predicates* that are defined in the conclusions of clauses with *abducible* (or *open*) *predicates* that occur in the conditions, but not in the conclusions of clauses. Abducible predicates are like extensional predicates in deductive databases, and closed predicates are like intensional predicates.

The classical application of abduction is to generate hypotheses, which are atomic sentences in the abducible predicates, to explain observations. The best known example of this is the use of abduction to explain the observation that *the grass is wet*, using the clauses:

the grass is wet if it rained.
the grass is wet if the sprinkler was on.

Abduction generates the two alternative hypotheses, *it rained* and *the sprinkler was on.* Assuming that the state of the environment is stored in a database, the hypotheses are alternative candidates for adding to the database. We will argue in the next section that deciding which of the alternatives to add is like conflict resolution in production systems.

ALP extends classical abduction in two ways: First, it not only generates abductive hypotheses to explain observations, but also it generates hypothetical actions to achieve goals; and second it constrains hypothetical explanations so that they do not violate integrity constraints.

Suppose that, in addition to the two clauses in the example above, we have:

Fact:	*the sun was shining.*
Integrity constraint:	*not (it rained and the sun was shining)*
or equivalently:	*if it rained and the sun was shining then false.*

Then the hypothesis *it rained* violates the integrity constraint, leaving the alternative hypothesis *the sprinkler was on* as the only acceptable explanation of the observation that *the grass is wet.*

> In general, given a logic program ***LP***, integrity constraints ***IC*** and a problem ***G***, which is either an achievement goal or an observation to be explained, a set Δ of atomic sentences in the abducible predicates is an *abductive explanation* or a *solution* of ***G*** if and only if both ***IC*** and ***G*** hold with respect to the extended logic program $\boldsymbol{LP} \cup \Delta$.

This characterisation of ALP is compatible with different semantics specifying what it means for a goal or integrity constraint to hold with respect to a logic program. It is also compatible with different proof procedures.

The most straight-forward proof procedures for ALP [15, 11, 20] are simple extensions of SLD or SLDNF in which the integrity constraints ***IC*** are represented as denials. The constraints ***IC*** and problem ***G*** are conjoined together and transformed into the form of a normal logic programming goal clause. The set *Δ* is constructed incrementally, adding new hypotheses to *Δ* to solve sub-goals in the abducible predicates that would otherwise fail. The incremental construction of *Δ* is interleaved with checking that the new hypotheses added to *Δ* satisfy the integrity constraints. Checking the integrity constraints may cause further updates to *Δ*.

For the sake of simplicity, these proof procedures avoid general-purpose integrity checking, and instead perform only one step of forward reasoning to match a newly added abductive hypothesis with a condition of an integrity constraint. For this purpose, the integrity constraints are preprocessed into a form in which at least one condition is abducible. The resulting preprocessed integrity constraints are similar to *event-condition-action* rules where the *event* and *action* predicates are abducible.

These approaches all use backward reasoning with SLD or SLDNF resolution, to reduce goals to sub-goals and to generate abductive hypotheses. An alternative approach, developed originally by Console, Dupre and Torasso [10] and extended by Fung and Kowalski [18] is to reason instead with the completions of the logic programming clauses defining the non-abducible predicates.

The completions in [28] all have the form of if-and-only-if (IFF) definitions:

atomic formula iff disjunction of conjunctions

where the *conjunctions* (also called *disjuncts*) are conjunctions of atomic formulae or conditionals. Negative literals *not p* are written as conditionals *if p then false*. The *atomic formula* is called the *head*, and the *disjunction of conjunctions* is called the *body* of the definition. Integrity constraints are represented as conditionals of the form:

if conditions then disjunction of conjunctions

where the *conditions* are a conjunction of atomic formulae and the conclusion has the same form as the body of a definition.

Because the conclusions of integrity constraints have the same form as the bodies of definitions, they can contain existentially quantified variables, and the quantification of the constraints can be implicit, as in the example:

If X is an employee then X has social security number Y.

Here X is universally quantified and Y is existentially quantified.

Because disjuncts in the bodies of definitions can contain conditionals, these conditionals can have the same form as integrity constraints with the same implicit quantification of variables, for example:

The banking department gets a 5 % bonus starting tomorrow iff
if X is an employee in the banking department today and X has salary S today
then X has salary S + .05S tomorrow.

Given an initial problem ***G*** and integrity constraints ***IC***, IFF conjoins ***G*** and ***IC*** in an initial goal, and rewrites it as a *disjunction of conjunctions*. Starting from the initial goal, equivalence-preserving inferences transform one state of the goal into another, equivalent state, also represented in the form of a *disjunction of conjunctions.* Each disjunct corresponds to a branch of the search space of abductive proof procedures based on SLD and SLDNF.

The inference rules of IFF include both backward and forward reasoning. Backward reasoning (also called *unfolding*) replaces an atomic formulae by the body of its definition. Forward reasoning (also called *propagation*) performs resolution between an atomic formula and a conditional in the same disjunct. The resolvent is added to the disjunct. An abductive explanation is generated, when a disjunct contains only atoms or negative literals in the abducible predicates (or equalities) and no further inferences can be performed within that disjunct.

The fact that every disjunct in the search space contains a copy of all the ICs is a potential source of inefficiency. This inefficiency can be avoided, simply by factoring out the ICs, representing them explicitly only once, but treating them as though they belong to every disjunct.

The IFF proof procedure was developed with the theorem-hood view of integrity satisfaction in mind. It is sound and, with certain modest restrictions on the form of clauses and integrity constraints, complete with respect to the completion of the program in the Kunen [33] three-valued semantics. The SLP proof procedure [32] is a refinement of IFF, in which integrity constraints are also used for constraint handling rules in constraint logic programming, but with the consistency view of integrity satisfaction. SLP is not complete in the general case, because consistency is not semi-decidable. CIFF [36], which is a successor of SLP, reverts to the theorem-hood semantics, and is complete in certain cases.

Compared with proof procedures that extend SLD or SLDNF, which represent integrity constraints as denials, the main attraction of the IFF proof procedure and its successors is that they include forward reasoning with integrity constraints represented explicitly as conditionals. Forward reasoning can also be used to derive consequences of abducible hypotheses, to help in choosing between them. This is important in ALP agents, as we will discuss in the next section.

Another useful feature of these proof procedures is that their representation of negative literals in the conditional form *if p then false* means that negative conditions can be unfolded even if they contain variables, and the IFF selection function, which is the analogue of the SLDNF selection function, can be much more liberal compared with SLDNF.

The completeness of IFF means that ICs can be satisfied, not only, like condition-action rules, by satisfying their conclusions when their conditions are satisfied, but also by making their conditions fail. For example, the integrity constraint *if you commit a mortal sin and don't go to confession then you will go to hell* can be satisfied either by not committing a mortal sin, committing a mortal sin and going to confession, or committing a mortal sin, not going to confession and going to hell. Here is a symbolic example:

G: *p*
IC: *if p and not q then a*

LP: *q if b*

where *p* is an observation, and *a* and *b* represent actions. IFF derives *not q then a* by forward reasoning, which it then rewrites as *q or a*. It unfolds *q* to obtain *b or a*. Thus the problem has two solutions, either do *b* or do *a*.

In the informally described abductive agent proof procedures of [23, 24], the forward reasoning integrity checking method of SK is combined with the IFF representation of integrity constraints as conditionals within an SLD-style framework. A variant of this is formalised in LPS [29]. Here is an example from [24, 26].

Integrity constraint: *If there is an emergency then I get help.*

Logic program:

A person gets help if the person alerts the driver.
A person alerts the driver if the person presses 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 suddenly ill.
There is an emergency if there is an accident.
There is a fire if there are flames.
There is a fire if there is smoke.

Abducible predicates: *there are flames, there is smoke, a person presses the alarm signal button*

Observation: *There is smoke.*
Forward reasoning: *There is a fire.*
Forward reasoning: *There is an emergency.*
Forward reasoning, Goal: *I get help*
Backward reasoning, Goal: *I alert the driver*
Backward reasoning, Solution: *I press the alarm signal button.*

Notice that the example integrates the behaviour of the three kinds of production rules identified in section 1.2. The first two steps of forward reasoning use clauses of the logic program as forward reasoning logic rules; the third step of forward reasoning uses the integrity constraint as a reactive rule, and the two steps of backward reasoning use clauses of the logic program as goal-reduction rules.

An alternative to using forward reasoning with logic programs in the first two steps in this example is to preprocess the integrity constraint (by unfolding the definitions of *there is an emergency* and *there is a fire*) into several separate integrity constraints:

If there are flames then I get help.
If there is smoke then I get help.
If one person attacks another then I get help.
If someone becomes suddenly ill then I get help.
If there is an accident then I get help.

The use of a single integrity constraint and forward reasoning with logic programs simulates forward chaining with production rules, and is arguably more natural. (Notice that these integrity constraints could be further pre-processed into condition-action rules, by unfolding the conclusion *I get help,* replacing it by the action *I press the alarm signal button.*)

Following the lead of other abductive proof procedures, the proof procedure illustrated in this example can be given a variety of logical semantics. To achieve our desired combination of logic programs, production systems and agents, it remains only to:

- represent actions by abducible predicates,
- embed the abductive proof procedure in an agent cycle, and
- justify the use of a destructively changing database of facts.

7 ALP agents

The notion of ALP agent, in which ALP is embedded as the thinking component of an observation-thought-decision-action cycle, was introduced in [22] and developed in [27, 28, 23, 24]. It is the basis of the KGP (Knowledge, Goals and Plans) agent model of [21]. The ALP agent proof procedure of [28] is the IFF proof procedure. The proof procedure of KGP is the CIFF proof procedure [36] which extends IFF with constraint handling procedures.

The observation-thought-decision-action cycle of an ALP agent is similar to other agent cycles. However, in ALP agents, an agent's beliefs are represented by logic programs, its goals are represented by integrity constraints, and its actions are represented by abducible predicates.

7.1 Observations

In the original ALP agent model, observations are added incrementally to the goals, and the IFF proof procedure generates explanations of observations as well as actions to achieve goals. However, in production systems and BDI agents, observations contribute to the current state of the database of facts. In the KGP agent model, observations are added to a separate database of facts, similar to the production system/BDI agent database. But whereas in PR/BDI systems the database represents only the current state of the environment, in KGP the database is a monotonically increasing representation of both the current state and all past states.

The ALP agent and KGP beliefs include the event calculus axioms [31] to derive new time-stamped facts from previous observations. However, the agent can bypass reasoning with the event calculus, by directly observing the environment instead. In both the ALP agent and KGP proof procedures, facts, whether directly observed or derived by the event calculus, are used both to solve atomic goals and to propagate with integrity constraints.

In both the ALP and the KGP agent models, in the routine use of the agent cycle, forward reasoning derives consequences of observations and triggers integrity constraints; and backward reasoning reduces goals to plans of action. Reasoning can

be interrupted both by incoming observations and by outgoing actions. An incoming observation, in particular, might trigger an integrity constraint and derive an action that needs to be performed immediately, interrupting the derivation of plans and the execution of actions that need to be performed only later in the future.

However, in more demanding situations, backward reasoning can also be used to generate explanations of observations, and forward reasoning can also be used to derive consequences of explanations, to help in choosing between alternative explanations. For example, if *it rained* and *the sprinkler was on* are alternative explanations for the observation *the grass is wet*, then forward reasoning might derive *the street is wet* from the hypothesis that *it rained*, and derive *the water meter reading is high* from the hypothesis that *the sprinkler was on*. Checking these consequences by attempting to observe them in the environment can help to prefer one explanation to the other.

A similar kind of reasoning forward from alternative candidate actions, to derive and evaluate their possible outcomes, can also help with decision-making and conflict resolution.

7.2 Decision-making and Conflict Resolution

The easy part of the extension of ALP to ALP agents is making ALP open to observations, and embedding ALP in a down-sized observation-thought cycle, in which the agent passively thinks about alternative plans of actions, but doesn't make any commitments to perform any actions. The hard part is to decide among the alternatives, and actually do something. To choose between different actions, an extended cycle can incorporate a decision making component, which generalises conflict resolution in production systems.

Consider the following "rules":

If someone attacks me then I attack them back
If someone attacks me then I run away

and suppose I observe *someone attacks me.* Treated as condition-action rules in a production system, both rules would be triggered, and conflict resolution would be needed to fire only one of them. However, in ALP and ALP agents, both rules and both conclusions *I attack them back* and *I run away* would need to be satisfied, which might not be possible. To avoid this problem and to have a logical semantics, the rules need to be re-formulated:

Maintenance goal: *If X attacks me then I protect myself against X*
Beliefs: *I protect myself against X if I attack X back*
I protect myself against X if I run away.

In ALP, given this re-formulation and the observation *someone attacks me*, it suffices to satisfy only one of *I attack them back* or *I run away*. But this still involves making a choice between the two alternatives, and then successfully performing one or more actions to make that choice succeed, and thereby to make the given instance of the goal become true. In the IFF syntax and proof procedure, the goal and the beliefs

could be pre-processed, by unfolding the conclusion *I protect myself* into the still simpler form of a single maintenance goal:

Maintenance goal: *If someone attacks me*
then (I attack them back or I run away).

Viewed in this way, conflict resolution in production systems is partly a compensation for the restricted syntax of production rules (no disjunction in conclusions of rules) and partly a form of Decision Theory. ALP agents do not suffer from the same restrictions on syntax; but, like production systems, they need to be augmented with a decision-making component.

In classical Decision Theory, actions are chosen to optimise expected outcomes. ALP can help with this, because it can be used, not only to generate alternative actions, but also to reason forward from candidate actions to derive their likely outcomes – for example to reason forward from the candidate action *we go for a picnic*, to derive the following outcome from the following belief:

Belief: *We will have a whale of a good time*
if we go for a picnic and the weather is good.
Outcome: *We will have a whale of a good time if the weather is good.*

To fit into a full-fledged decision-making procedure, outcomes (e.g. *We will have a whale of a good time*) need to be evaluated for their utility, and abducible conditions beyond the agent's control (e.g. *the weather is good*) need to be assigned a probability. ALP is particularly well-suited to this, as Poole [39, 40] has shown, because it is very natural and very easy to associate probabilities with abducible predicates. The resulting Decision Theoretic analysis, or some computationally less expensive approximation to it, fits comfortably into an ALP agent cycle [24, 38]

Having done the analysis and made the decision, the agent must still commit to performing an action. The action may be part of a plan; and even if the action succeeds the plan might fail, because for some reason or other the agent might not be able to perform the other actions in the plan. For example the preconditions of later actions might not hold or the actions might get timed out. The upshot of all these complications is that it may not be possible to commit to only one alternative plan for achieving a higher level goal. It might be necessary to embark upon one plan, and then switch to another plan if the first plan fails. It may also be necessary to re-perform an action in the same plan. The ALP agent cycle allows all these options.

7.3 Semantics

The semantics of ALP agents can be understood in ALP terms. The fact that observations and actions occur in a temporal order can be dealt with simply by including the time or state of observations and actions as explicit parameters of predicates. The effect of executing actions can be taken into account by including an action theory, such as the situation calculus or event calculus in the agent's set of beliefs ***B***. With these assumptions, and the further assumption that the observations do

not include any post-conditions (effects) of the agent's own actions, the semantics of the ALP agent cycle is a special case of the ALP semantics.

> Given beliefs $\boldsymbol{B}$, goals $\boldsymbol{G}$, initial database state S_0 and possibly an infinite set $\boldsymbol{O} = \{O_1, O_2, \ldots, O_n, \ldots\}$ of input (external) observations, an *ALP agent solution* is a possibly infinite set $\Delta = \{A_1, A_2, \ldots, A_m, \ldots\}$ of actions such that $\boldsymbol{G}$ is satisfied by the logic program $\boldsymbol{B} \cup \boldsymbol{O} \cup \Delta$.

Theorem 7.1 of [28] establishes a correspondence between the static behaviour of the IFF proof procedure and the dynamic behaviour of ALP agents with observations. The soundness of the ALP agent cycle follows from that theorem.

8 The Representation of State Change and the Frame Problem

For the ALP agent semantics to work, observations and actions need to have an explicit representation of time or state. This gives rise to the *frame problem* of how to represent and reason about change of state correctly and efficiency. It is generally held that the frame problem can be solved by means of an appropriate action theory such as situation calculus or event calculus. These and similar calculi all include some form of frame axiom, such as:

fact F holds in state S+1 if fact F holds in state S
and S+1 is obtained from S by action/event A
and A does not terminate F.

Given an appropriate semantics, including the various semantics developed for logic programming, it has been argued that these formulations of the frame axiom solve the frame problem.

However, all of these solutions reason explicitly, whether forward or backward, that a fact holds in a state *S+1* if it held in state *S* and was not terminated in the state transition. Reasoning in the forward direction, for example, every unterminated fact holding in state *S* needs to be explicitly copied–to the new state *S+1*. Moreover, without sophisticated optimisations and garbage collection, the same unterminated facts need to be duplicated in both states *S* and *S+1*, and indeed in all states from the initial state to the current state. Backward reasoning does not store unterminated facts redundantly, but requires a potentially expensive calculation instead. Thus it can be argued that that aspect of the frame problem that is concerned with efficiently reasoning about change of state has no solution within a purely logical representation.

It is probably for such reasons of efficiency that production systems and BDI agents store only the current state, use destructive assignment to generate a new current state from the previous state, and do not employ an explicit representation of states or time at all.

It can be argued that the ALP agent model with its event calculus representation of change also suffers from the inefficiencies associated with the frame problem. However, in ALP agents, to determine whether a fact holds in the current state, these inefficiencies can be avoided, by directly observing whether the fact holds in the environment instead. In such a case, the environment serves as an auxiliary, external

database, which contains a complete record of the current state. To test these ideas, we have developed LPS (Logic-based Production System) [29], which is a variant of ALP agents without frame axioms, but with an operational semantics that incorporates a destructively changing database. For simplicity, as is common in production systems, we assume that there are no external observations, once the initial state of the database has been given as input.

The LPS operational semantics maintains the current state of a deductive database, with facts (atomic sentences) defining the extensional predicates and logic programs (or deduction rules) defining intensional predicates. The actions that change the current state affect only the extensional predicates, which are represented without explicit state parameters. As a consequence, to execute an action and to change the current state, it suffices to delete the facts terminated by the action and to add the facts initiated by the action. Facts that are not affected by the action persist without the need to reason that they persist, simply because they do not change. Intentional predicates change implicitly as ramifications of changes to the extensional predicates.

In contrast with the operational semantics, which maintains only the current state, the LPS declarative semantics is based on the sequence of database states with the implicit state parameter of the operational semantics made explicit. The semantics also requires that the sequence of states conforms to a logical representation of change, such as the event calculus *EC* represented in logic programming form:

> Given beliefs ***B*** in the form of a logic program, goals ***G*** in the form of a set of integrity constraints, and an initial database state S_0, a possibly infinite set $\Delta = \{A_1, A_2 ,\ldots, A_m ,\ldots\}$ of actions is an *LPS solution* if and only if ***G*** is satisfied by the logic program $\boldsymbol{B} \cup S_0 \cup \Delta \cup EC$.

As shown in [29], LPS is sound, but is not complete for the same reason that condition-action rules are not complete (because they cannot make their conditions false).

9 Conclusions

We have argued for combining the functionalities of production systems and logic programs, while retaining their individual contributions and eliminating their overlap. To defend our thesis that the two kinds of rules provide distinct functionalities and need to be combined, we have appealed to the distinctions

- in psychology between descriptive and deontic conditionals,
- in intelligent agents between beliefs and goals,
- in deductive databases between deduction rules and integrity constraints, and
- in ALP between logic programs and integrity constraints.

We have advocated an approach that combines the two kinds of rules in an observation-thought-decision-action cycle, along the lines of ALP agents, and sketched an approach that combines a declarative, logic-based semantics with an

operational semantics that operates with the current state of a destructively changing database.

References

1. Alferes, J.J., Banti, F., Brogi A.: An Event-Condition-Action Logic Programming Language. In: 10th European Conference on Logics in Artificial Intelligence, M. Fisher, W. van der Hoek, B. Konev and A. Lisitsa (eds.), JELIA06: Lecture Notes in Artificial Intelligence 4160, pp 29--42 Springer-Verlag (2006).
2. Alferes, J. J., Brogi, A., Leite, J. A., Pereira, L. M. : Evolving Logic Programs. In: 8th European Conference on Logics in Artificial Intelligence (JELIA'02), S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), pp. 50--61, Spriger-Verlag, LNCS 2424, Springer-Verlag (2002)
3. Anderson, J., Bower, G.: Human Associative Memory. Washington, D.C.: Winston (1973)
4. Bratman, M. E., Israel, D. J., Pollack M. E.: Plans and Resource-bounded Practical Reasoning. Computational Intelligence, 4, 349--355 (1988)
5. Caroprese, L. Greco, S., Sirangelo, C., Zumpano, E. : Declarative Semantics of Production Rules for Integrity Maintenance. In: 22nd International Conference on Logic Programming, Etalle, S., Truszczynski, M. (eds.), LNCS 4079, pp. 26--40 (2006)
6. Ceri S., Fraternali P., Paraboschi S., Tanca L.: Automatic Generation of Production Rules for Integrity Maintenance. ACM Transactions on Database Systems (TODS), Volume 19, Issue 3 (September 1994), 367--422 (1994)
7. Ceri, S., Widom, J.: Deriving production rules for constant maintenance. In: 16th International Conference on Very Large Data Bases, pp. 566-577 (1990)
8. Cheng P.W., Holyoak, and K. J. : Pragmatic Reasoning Schemas. Cognitive Psychology, 17, 391--416 (1985)
9. Clark, K. : Negation as Failure. In: Readings in Nonmonotonic Reasoning, Morgan Kaufmann, pp. 311--325 (1978)
10. Console, L., Theseider Dupre, D., Torasso, P.: On the Relationship Between Abduction and Deduction. Journal of Logic and Computation 1(5), 66--690 (1991)
11. Denecker, M., De Schreye, D.: SLDNFA: An Abductive Procedure for Normal Abductive Programs. J. Logic Programming, 34(2), pp. 111--167 (1998)
12. Dennis, L.A., Bordini, R.H., Farwer, B., Fisher, M.: A Common Semantic Basis for BDI Languages. In: 5th International Workshop on Programming Multi-Agent Systems (PROMAS), pp. 124--139 (2007)
13. d'Inverno, M., Luck, M., Georgeff, M. P., Kinny, D., Wooldridge, M. : A Formal Specification of dMARS. In: Intelligent Agents IV: 4th International Workshop on Agent Theories, Architectures and Languages. Lecture Notes in Artificial Intelligence, 1365, Springer-Verlag, pp. 155--176 (1998)
14. Dung, P.M., Mancarella, P. : Production Systems with Negation as Failure. IEEE Transactions on Knowledge and Data Engineering, Volume 14 , Issue 2, 336--352 (2002)
15. Eshghi, K., Kowalski, R.: Abduction Compared with Negation by Failure. In: 6th International Conference on Logic Programming, G. Levi and M. Martelli (eds.) MIT Press, pp. 234--254 (1989)
16. Fernandes, A.A.A., Williams, M.H., Paton, N.: A Logic-Based Integration of Active and Deductive Databases. New Generation Computing, Volume 15, Number 2, 205--244 (1997)
17. Fraternali P., Tanca L.: A Structured Approach for the Definition of the Semantics of Active Databases. ACM Transactions on Database Systems (TODS) Volume 20, Issue 4 (December 1995), 414--471 (1995)
18. Fung, T.H., Kowalski, R.: The IFF Proof Procedure for Abductive Logic Programming. Journal of Logic Programming, 33 (2), 151--164 (1997)

19. Kakas, A., Kowalski, R., Toni, F. : The Role of Logic Programming in Abduction. In: Gabbay, D., Hogger, C.J., Robinson, J.A. (eds.): Handbook of Logic in Artificial Intelligence and Programming 5, Oxford University Press, pp. 235--324 (1998)
20. Kakas, A.C., Mancarella, P.: Abduction and Abductive Logic Programming. In: 11th International Conference on Logic Programming, pp. 18--19 (1994)
21. Kakas, A., Mancarella, P., Sadri, S., Stathis, K. and Toni, F.: Computational Logic Foundations of KGP Agents. Journal of Artificial Intelligence Research (2009)
22. Kowalski, R. : Using Metalogic to Reconcile Reactive with Rational Agents. In: K. Apt and F. Turini (eds.), Meta-Logics and Logic Programming, MIT Press (1995)
23. Kowalski, R. : Artificial Intelligence and the Natural World. Cognitive Processing, 4, 547--573 (2001)
24. Kowalski, R.: The Logical Way to be Artificially Intelligent. In: 6th CLIMA, F. Toni and P. Torroni (eds.), Springer Verlag, LNAI, pp. 1--22 (2006)
25. Kowalski, R.: Reasoning with Conditionals in Artificial Intelligence. In: M. Oaksford (ed.), The Psychology of Conditionals, Oxford University Press, to appear (2009)
26. Kowalski, R.: How to be Artificially Intelligence. To be published, Cambridge University Press (2010) Draft on http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html
27. Kowalski, R., Sadri, F.: Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In: International Workshop on Logic in Databases, San Miniato, Italy, Springer-Verlag, LNCS 1154, pp. 131--150 (1996)
28. Kowalski, R., Sadri, F.: From Logic Programming towards Multi-agent Systems. Annals of Mathematics and Artificial Intelligence, 25, 391--419 (1999)
29. Kowalski, R., Sadri, F.: LPS - a Logic-Based Production System Framework. Department of Computing, Imperial College (2009)
30. Kowalski, R., Sadri, F., Soper, P.: Integrity Checking in Deductive Databases. In: 13th VLDB, Morgan Kaufmann, Los Altos, Ca., pp. 61--69 (1987)
31. Kowalski, R., Sergot, M.: A Logic-based Calculus of Events. In: New Generation Computing, Vol. 4, No.1, 67--95 (1986). Also in: Inderjeet Mani, J. Pustejovsky, and R. Gaizauskas (eds.), The Language of Time: A Reader, Oxford University Press (2005)
32. Kowalski, R., Toni, F., Wetzel, G.: Executing Suspended Logic Programs. Fundamenta Informatica 34 (3), 1--22 (1998)
33. Kunen, K.: Negation in Logic Programming. Journal of Logic Programming, 4:4 289--308 (1987)
34. Laird, J.E., Newell, A., Rosenblum, P. S.: SOAR: an Architecture for General Intelligence. Artificial Intelligence, 33:1, 1--64 (1987)
35. Lloyd, J.W., Topor, R.W.: A Basis for Deductive Database Systems. J. Logic Programming 2: 93--109 (1985)
36. Mancarella, P., Terreni, G., Sadri, F., Toni, F., Endriss, U.: The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Theory, Implementation and Experiments. Theory and Parctice of Logic Programming, to appear (2009)
37. Nicolas, J.M., Gallaire, H.: Database: Theory vs. Interpretation. In: Gallaire, H., Minker, J. (eds.), Logic and Databases, Plenum, New York (1978)
38. Pereira, L. M. , Anh H. T.: Evolution Prospection. In: First KES Intl. Symposium on Intelligent Decision Technologies (KES-IDT'09), April 2009, K. Nakamatsu et al. (eds.), Springer Studies in Computational Intelligence, 199, Himeji, Japan, pp. 51--64 (2009)
39. Poole, D.: Probabilistic Horn Abduction and Bayesian Networks. Artificial Intelligence, 64(1), 81--129 (1993)
40. Poole, D.: The Independent Choice Logic for Modeling Multiple Agents Under Uncertainty. Artificial Intelligence, Vol. 94, 7--56 (1997)
41. Rao, A.S.: Agents Breaking Away. Lecture Notes in Artificial Intelligence, Volume1038, Netherlands (1996)

42. Rao, A. S., Georgeff, M. P.: Modeling Rational Agents with a BDI-Architecture. In: 2nd International Conference on Principles of Knowledge Representation and Reasoning, pp. 473--484 (1991)
43. Reiter, R.: On Closed World Data Bases. In: Gallaire H. and Minker J. (eds.), Logic and Data Bases, Plenum Press, New York, pp. 55-76 (1978)
44. Reiter, R.: On Integrity Constraints. In: 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, pp. 97--111 (1988)
45. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (2nd ed.). Upper Saddle River, NJ: Prentice Hall (2003)
46. Sadri, F.: A Theorem-Proving Approach to Database Integrity. PhD Thesis, Imperial College, (1988)
47. Sadri F., Kowalski R.: A Theorem-Proving Approach to Database Integrity. In: Minker, J. [ed.], Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 313-362 (1988)
48. Simon, H.: Production Systems. In Wilson, R. and Keil, F. (eds.): The MIT Encyclopedia of the Cognitive Sciences, The MIT Press, 676-677 (1999)
49. Stenning, K., van Lambalgen M.: Human Reasoning and Cognitive Science. MIT Press (2008)
50. Thagard, P.: Mind: Introduction to Cognitive Science. Second Edition. MIT Press (2005)
51. Wason, P. C.: Reasoning About a Rule. The Quarterly Journal of Experimental Psychology, 20:3, 273--281 (1968)
52. Raschid, L. : A Semantics for a Class of Stratified Production System Programs. J. Log. Program. 21(1): 31--57 (1994)
53. Lausen, G., Ludäscher, B., May, W.: On Active Deductive Databases: The Statelog Approach. In: Transactions and Change in Logic Databases, Decker, H., Freitag B., Kifer, M., Voronkov, A. (eds.), LNCS 1472, Springer (1998)
54. Zaniolo, C.: On the Unification of Active Databases and Deductive databases. In: 11th British National Conference on Databases, pp. 23-39 (1993)