Production Systems Need Negation As Failure

Phan Minh Dung
Computer Science Program, Asian Institute of Technology
PO Box 2754, Bangkok 10501, Thailand
dung@cs.ait.ac.th

Paolo Mancarella
Dipartimento di Informatica, University of Pisa
Corso Italia 40, 56125 Pisa, Italy
paolo@di.unipi.it

Abstract

We study action rule based systems with two forms of negation, namely classical negation and "negation as failure to find a course of actions". We show by several examples that adding negation as failure to such systems increase their expressiveness, in the sense that real life problems can be represented in a natural and simple way. Then, we address the problem of providing a formal declarative semantics to these extended systems, by adopting an argumentation based approach, which has been shown to be a simple unifying framework for understanding the declarative semantics of various nonmonotonic formalisms. In this way, we naturally define the grounded (well-founded), stable and preferred semantics for production systems with negation as failure. Next, we characterize the class of stratified production systems, which enjoy the properties that the above mentioned semantics coincide and that negation as failure can be computed by a simple bottom-up operator.

Introduction and Motivations

In this section we first give examples to motivate the extension of the production systems paradigm (Hayes-Roth 1985) by the introduction of negation as failure (to find a course of actions). We then discuss its role as a specification mechanism for reactive systems.

On the need for negation as failure in production systems

Example 1 Imagine the situation of a person doing his household work. Clothes have to be washed and the person has two options, either hand washing or machine washing. If there is machine powder in house, then machine washing can take place. This is represented by the production rule

r_1 : **if** Powder **then** machine-wash.

If no machine powder is in house, then it can be acquired by either buying it in the shop (provided the shops are open) or by borrowing it from the neighbor (if he is in). The rules for acquiring powder can be represented by the following two classical production rules

r_2 : **if** $\neg$ Powder, Shop-Open **then** buy
r_3 : **if** $\neg$ Powder, Neighbor-In **then** borrow.

Of course, hand washing is undesirable and will be taken up if *there is no way to acquire machine powder*. The naive representation of this rule using classical negation

if $\neg$ Powder **then** hand-wash

is clearly not correct, since the meaning of such a rule is that if there is no machine powder in house at the current state, then the clothes should be hand washed, while the intuitive meaning of "there is no way to acquire machine powder" is that there is no *course of actions* starting from the current state leading to acquiring machine powder. Hence in a state where there is no machine powder in house and the neighbor is in, the above naive representation would allow hand washing though there is a way to acquire machine powder by borrowing it from the neighbor. Hence it fails to capture the intuitive understanding of the problem.

Here we need to use a different kind of negation, called negation as failure (to find a course of actions) and denoted by the operator *not*. The previous naive representation is now replaced by

r_4 : **if** *not* Powder **then** hand-wash.

It is not difficult to find other real life situations governed by rules with negation as failure.

Example 2 Consider the rules for reviewing the work of faculties at the end of each academic year in a university. The first rule specifies the conditions for offering tenure to assistant professors. It states that assistant professors with good publications and with a working experience of at least five years should be offered tenure. This rule could be formalized by:

if Assistant-Prof(X), Good-Pub(X),
Work-at-least-5-years(X) **then** offer-tenure(X).

The second rule states that *if an assistant professor has no prospect of getting a tenure then fire him.* Though the intuitive meaning of this rule is clear, it is not possible to represent it as a classical production rule since the premises of a classical production rule represent conditions which must be satisfied in the current state of the world while the premises of the second rule represent a projection into the future. It says that if there is no possibility for an assistant professor to get a tenure in the future then sack him now. In other words, the rule says that if an assistant will fail in all possible course of actions in the future to get a tenure then fire him. To represent this rule, we use again negation as failure to find a course of actions. The second rule can then be represented as follows:

if Assistant-Prof(X), *not* Getting-Tenure(X)
then fire(X)

In real life, we often find ourselves in situations where we have to deal with risky or undesirable actions. For example, a doctor may have to take the decision of cutting the foot of his patient due to some severe frostbite. This is a very risky, undesirable decision and the commonsense rule specifying the conditions for taking this action is that the doctor is allowed to cut if *there is no other way* to save the foot of the patient. This can be represented, using negation as failure, as follows:

if *not* Save **then** cut.

Finally, we can expect that in real life, intelligent systems could be employed to satisfy multiple goals. These goals can have different priorities, and negation as failure (to find a course of actions) can be used to represent these priorities as in the following example.

Example 3 Consider a robot fire fighter that should be sent into a fire to save lives and properties. The priority here is certainly saving lives first. Imagine now that the robot is standing before a valuable artifact. Should it take it and get out of the fire? The answer should be yes *only if* the robot is certain that there is nothing it can do to save any life. The rule can be represented as follows

if Artifact(X), In-Danger(X), *not* Human-Found
then save(X)

Note that the *not* Human-Found here means that no human being could be found in the current and all other possible states of the world reachable by firing a sequence of actions the robot is enabled to perform.

Negation as failure as a specification mechanism for reactive systems

Let us consider again the example 1. Checking whether the conditions of the rule

r_4 : **if** *not* Powder **then** hand-wash

are satisfied in the current state involves checking whether there is any way to acquire machine powder from the current state, a process which could be time consuming and expensive. In a concrete application, as in our example where there are only two ways to get powder: buying it in the shop or borrowing it from the neighbor, negation as failure can be "compiled" into classical negation to produce a more efficient rule:

if $\neg$ Powder, $\neg$ Shop-Open, $\neg$ Neighbor-In
then hand-wash.

However, the environment in which a production system with the above rule is applied can change. For example, you may get a new neighbor who may not have any interest for good relations to other peoples, and so you will not be able to borrow anything from him. Hence the rule for borrowing must be dropped. Consequently the above production rule must be revised to

if $\neg$ Powder, $\neg$ Shop-Open **then** hand-wash

It is clear that the rule r_4 with negation as failure is still correct and serves as a specification for checking the correctness of the new rule.

The point we want to make here is that in many cases, though negation as failure is not employed directly, it could be used as a specification mechanism for a classical production system. This situation can be encountered quite often in many real life situations. Imagine the work of a physician in an emergency case dealing with a patient who is severely injured in a road accident. In such cases, where time is crucial, what a doctor would do is to follow certain treatments he has been taught to apply in such situations. He more or less simply *react* depending on the physical conditions of the patient. The treatment may even suggest a fateful decision to operate the patient to cut some of his organs. Now it is clear that such treatment changes according to the progress of the medical science. One treatment which was correct yesterday may be wrong today. So what decides the correctness of such treatments? We can think of such treatments in a simplified way as a set of production rules telling the doctors what to do in a concrete state of a patient. The correctness of such rules are determined by such *commonsense principles* like: *Operate and cut an organ only if there is no other way to save the patient.* And such a principle can be expressed using negation as failure.

Aim of this work

We have seen in the examples that using negation as failure in production systems allows one to naturally and correctly represent many real life problems. The main aim of this paper is to provide a declarative semantics to production systems where two kinds of negation are used, classical negation and negation as failure (to find a course of actions). In this respect, we show that the argumentation based approach (Dung 1995), which has been successfully adopted to understand logic programming with negation as failure as well as many other nonmonotonic formalisms, can also be adopted to provide a natural and simple declarative semantics to production systems with two kind of negations. The basic idea is that negation as failure literals, such as "*not* Powder" in example 1, represent *assumptions* underlying potential computations of a production system. The intuitive meaning of such an assumption is that the computation goes on by assuming that there is no course of actions (i.e. computation) from the current state of the world leading to a state which defeats the assumption itself. Referring back to example 1, assuming *not* Powder corresponds to assuming that from the current state there is no course of actions leading to a state where machine powder is in house. A computation which is supported by a sequence of assumptions is plausible (acceptable) if its underlying assumptions cannot be defeated by actually find a course of actions which defeats them.

These informal, intuitive notions can be formalized by viewing a production system as an argumentation system along the lines of (Dung 1995). This provides us with many natural semantics, such as the grounded (well-founded), the preferred and the stable semantics (Dung 1995). These semantics are arguably the most popular and widely accepted semantics for nonmonotonic and commonsense reasoning in the literature (Gelfond & Lifschitz 1988; Bondarenko *et al.* 1995; McDermott & Doyle 1980; Van Gelder, Ross, & Schlipf 1988).

Moreover, we address the problem of actually computing negation as failure. In this respect, we introduce the class of *stratified* production systems, where negation as failure can be computed using a simple bottom-up operator. As for the case of general stratified argumentation systems, stratified production systems enjoy the property that all the previously mentioned semantics (grounded, preferred and stable) coincide.

Classical production systems

We introduce here the notations and basic terminologies we are going to use in the following. The production system language we use is similar to classical ones (see, e.g. (Forgy 1981)). We assume a first order language $\mathcal{L}$ representing the ontology used to describe the domain of interest. A *state* of the world is interpreted as a snapshot of this world, hence is represented as a Herbrand interpretation of $\mathcal{L}$, i.e. as a set of ground atoms of $\mathcal{L}$. The set of states is denoted by *Stat*. Further we assume that a set of *primitive actions* $\mathcal{A}$ is given. The semantics (effect) of actions is described by the function

$$effect : \mathcal{A} \times Stat \longrightarrow Stat.$$

A *production rule* is a rule of the form

if $l_1, \ldots, l_n$ **then** a

where $l_1, \ldots, l_n$ are ground literals of $\mathcal{L}$ and a is an action in $\mathcal{A}$. The conditions (resp. action) of a rule r will be referred to by $cond(r)$ (resp. $action(r)$). A *production system* P is a set of production rules.

A production rule **if** $l_1, \ldots, l_n$ **then** a is said to be *applicable* in a state S iff the conditions $l_1, \ldots, l_n$ are true in S, i.e. $S \models l_1 \wedge \ldots \wedge l_n$.

Definition 4 A *partial computation* C of a production system P is a sequence

$$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_n} S_n$$

$n \geq 0$, such that S_i's are states, r_i's are production rules in P and for each $i \geq 1$, r_i is applicable in S_{i-1}, and $S_i = effect(action(r_i), S_{i-1})$. S_0 will be referred to as $initial(C)$ and S_n as $final(C)$.

A partial computation C

$$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_n} S_n$$

is called a *complete computation* if no production rule in P is applicable in S_n. ◇

Note that, if $n = 0$, then the partial computation is an empty computation.

The behavior of a production system P can be defined as the set of pairs of states (S, S') such that S (resp. S') is the initial (resp. final) state of some complete computation of P. This is formalized in the next definition.

Definition 5 For a production system P, the input-output semantics of P is defined by

$$\mathcal{IO}(P) = \{(initial(C), final(C)) \mid C \text{ is a complete computation of } P\}.$$ ◇

Production rules with NAF

We introduce a new form of negation into the language $\mathcal{L}$, denoted by *not*. A general literal is now either a (classical) literal l or a *naf*-literal *not* l, where l is a classical literal. For each classical literal l, the intuition of *not* l is that *it is not possible to find a course of actions to achieve* l.

Definition 6 A *general* production rule has the form

if $l_1, \ldots, l_n$ **then** a_i

where each l_i is a ground general literal. ◇

Given a general production rule

if $l_1, \ldots, l_n$ **then** a_i

the set of classical literals in r will be referred to as $cl\text{-}cond(r)$, and the set of naf-literals will be referred to as $hyp(r)$.

A *general* production system (GPS) P is a set of general production rules. A general production rule is said to be *possibly applicable* in a state S if $S \models cl\text{-}cond(r)$. Notice that a rule satisfying the condition that $S \models cl\text{-}cond(r)$ in a state S is not necessarily applicable in S since it is not clear whether its naf-conditions are satisfied in S.

Definition 7 Given a GPS P, a *possible partial computation* in P is a sequence

$$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_n} S_n.$$

where S_i's are states, r_i's are general production rules in P, for each $i \geq 1$, r_i is possibly applicable in S_{i-1} and $S_i = effect(action(r_i), S_{i-1})$.

Given a GPS P, the set of all possible partial computations of P is denoted by $\mathcal{C}(P)$. ◇

When a possible partial computation is acceptable: a motivating discussion

The basic idea in understanding the meaning of naf-conditions *not* l's in general production rules is to view them as hypotheses which can be assumed if there is no possible course of actions to achieve l. So, intuitively we can say that a rule is applicable in a state S if it is possibly applicable and each of its hypotheses could be assumed. A partial (pre)computation is then an acceptable partial computation if each of its rules is applicable. The whole problem here is to understand formally what does it mean that there is no possible course of actions starting from a state S to achieve some result l. Let us consider again the example 1.

Example 8 Let us consider again the washing example. The effects of the actions are specified below:

$effect(\text{hand-wash}, S) = S \cup \{\text{Clean}, \text{Tired}\}$
$effect(\text{machine-wash}, S) = (S \setminus \{\text{Powder}\}) \cup \{\text{Clean}\}$ if Powder $\in S$
$effect(\text{machine-wash}, S) = S$ if Powder $\notin S$
$effect(\text{buy}, S) = S \cup \{\text{Powder}, \text{Less-Money}\}$
$effect(\text{borrow}, S) = S \cup \{\text{Powder}\}$

Assume that in the initial state we have no powder, shops are closed and the neighbor is in. This state is represented by the interpretation $S_0 = \{\text{Neighbor-In}\}$. From this state there are three possible nonempty partial computations starting from S_0, namely

$C_1 : S_0 \xrightarrow{r_4} \{\text{Neighbor-In, Clean, Tired}\}$
$C_2 : S_0 \xrightarrow{r_3} \{\text{Neighbor-In, Powder}\} \xrightarrow{r_1} \{\text{Neighbor-In, Clean}\}.$
$C_3 : S_0 \xrightarrow{r_3} \{\text{Neighbor-In, Powder}\}$

First, notice that both C_2 and C_3 are not based on any assumption. Our commonsense dictate that C_2 and C_3 represent acceptable course of actions from the initial state which lead to the commonsense result that clothes are machine washed. Hence they both must be accepted as possible courses of actions. On the other hand C_1 is based on the assumption "*not* Powder", meaning that C_1 assumes that there is no possible way to acquire the machine powder. However, C_3 represents just one such possible way. Hence, C_3 represents an *attack* against the assumption "*not* Powder". So C_3 can also be viewed as an attack against the acceptability of C_1 as a legitimate computation. On the other hand, both C_2 and C_3 are not based on any assumption, hence there is no way they can be attacked.

This example points out that the semantics of GPS's is a form of argumentation reasoning, where arguments are represented by possible partial computations. In the following, we first recall the general notion of argumentation systems from (Dung 1995) and then we show that the natural semantics of GPS can be defined using the theory of argumentation.

Argumentation systems We review here the basic notions and definitions of argumentation systems (the reader can refer to (Dung 1995) for more details and for a discussion of the role of argumentation systems in many fields of Artificial Intelligence).

An *argumentation system* is a pair $\langle AR, attacks \rangle$ where AR is the set of all possible *arguments* and $attacks \subseteq AR \times AR$, representing the attack relationship between arguments. If the pair $(A, B) \in attacks$, then we say that A attacks B or B is attacked by A. Moreover, A attacks a set of arguments H if A attacks an argument $B \in H$. We also say that H attacks A if an argument $B \in H$ attacks A.

A set H of arguments is *conflict-free* if no argument in H attacks H.

An argument A is defended by a set of arguments H if H attacks any attack against A, i.e. for each argument $B \in AR$, if B attacks A then H attacks B. We also say that H defends A if A is defended by H.

The basic notion which underlies all the semantics for argumentation systems that we are going to review in the rest of this section, is the following, intuitive notion of acceptability of a set of arguments. A set H of arguments is *acceptable* if it is conflict-free and it can defend each argument in it.

Let H be a set of arguments and let $Def(H)$ be the set of all arguments which are defended by H. It is not difficult to see that H is acceptable iff $H \subseteq Def(H)$ and H is conflict-free. Further it is easy to see that $Def : \mathcal{P}(AR) \rightarrow \mathcal{P}(AR)$ is monotonic. Hence the equation $H = Def(H)$ has a least solution which is also acceptable (following from the fact that $Def(\emptyset)$ is acceptable and if H is acceptable then also $H \cup Def(H)$ is also acceptable).

The various semantics for argumentation systems are basically solutions of the above equation $H = Def(H)$. In particular, the *grounded* (well-founded) semantics of an argumentation system is the least solution of the equation $H = Def(H)$.

Another semantics for argumentation systems, called the *preferred semantics*, is defined by the maximal acceptable sets of arguments. It is not difficult to see that these sets are the conflict-free maximal solutions of the equation $H = Def(H)$. In general, preferred sets contain the grounded semantics, but do not coincide with it. In the next section, we will give an example for this.

Finally, a popular semantics of nonmonotonic reasoning and argumentation systems is the stable semantics, defined as follows. A conflict-free set of arguments H is said to be *stable* if it attacks each argument not belonging to it. It is not difficult to see that each stable set of arguments is acceptable. Furthermore, it is also easy to see that each stable set is preferred, hence it is a maximal, conflict-free solution of the equation $H = Def(H)$, but not vice versa.

It has been shown that argumentation systems provide a simple and unifying semantical framework for commonsense reasoning. For example, logic programming and different logics for nonmonotonic reasoning and n-person games are showed to be different representations of the argumentation systems presented above (see (Dung 1995) for further details) where for instance, the grounded semantics of argumentation corresponds to the well-founded semantics in logic programming (Van Gelder, Ross, & Schlipf 1988) and stable semantics of argumentation correspond to stable semantics of logic programming (Gelfond & Lifschitz 1988) and other prominent nonmonotonic logics like Reiter's default logic (Reiter 1980) or Moore's autoepistemic logic (Moore 1985).

In the following we show that general production systems with two kinds of negation are also a form of argumentation systems.

Computations as arguments The semantics of a GPS P is defined by viewing it as an argumentation framework $\langle AR(P), \textit{attacks}\rangle$, where $AR(P)$ is the set of all possible partial computations of P and the relation *attacks* is defined as follows.

Definition 9 Let C be a possible partial computation
$S_0 \xrightarrow{r_1} S_1 \longrightarrow \ldots \longrightarrow S_i \xrightarrow{r_{i+1}} S_{i+1} \longrightarrow \ldots \xrightarrow{r_n} S_n$.
An *attack* against C is a possible partial computation C' such that $\textit{initial}(C') = S_i$, for some i, and there exists an underlying assumption *not* l in $\textit{hyp}(r_{i+1})$ such that l holds in $\textit{final}(C')$. ◇

Remark: Empty computations cannot be attacked. Hence, empty computations are contained in any semantics.

Notice that, in the above definition, the initial state of the attack C', which defeats the assumption *not* l underlying C, has to be the actual state S_i in which such an assumption was made. In other words, whether an assumption *not* l can be defeated or not, depends on the state in which this assumption is made and on whether or not this state can be lead by a computation to a state in which l holds.

The view of a GPS as an argumentation system, allows us to provide it with three different semantics: grounded (well-founded), preferred and stable semantics.

It is easy to see that the following proposition hold.

Proposition 10
(i) If a computation C attacks a computation C' and C' is a prefix of C'', then C also attacks C''.
(ii) Let H be a set of computations which is either the grounded, or a preferred, or a stable set. For any $C \in H$, any prefix of C also belongs to H. ◇

We can now define the set of complete acceptable computations, with respect to a selected semantics.

Definition 11 Let P be a GPS and $\mathcal{R}$ be a selected semantics of P, i.e. $\mathcal{R}$ is either the grounded, or a preferred, or a stable, set of possible partial computations. A partial computation $C \in \mathcal{R}$ is called an *$\mathcal{R}$-complete* computation if there exists no other partial computation $C' \in \mathcal{R}$ such that C is a prefix of C'. ◇

If all the semantics of a GPS coincide, we simply talk about *complete* computations instead of $\mathcal{R}$-complete computations. Referring back to the example 8, the only complete computation starting from S_0 is C_2.

The input-output semantics of classical production systems can be extended to general production systems with respect to a selected (grounded, preferred, stable) semantics.

Definition 12 Let P be a GPS.
(i) The grounded input-output semantics of P is defined by
$\mathcal{IO}_G(P) = \{(\textit{initial}(C), \textit{final}(C)) \mid C \text{ is a grounded complete computation}\}$.
(ii) Let $\mathcal{R}$ be a set of arguments which is preferred or stable. Then
$\mathcal{IO}_{\mathcal{R}}(P) = \{(\textit{initial}(C), \textit{final}(C)) \mid C \text{ is an } \mathcal{R}\text{-complete computation of } P\ \}$. ◇

Stratified Production Systems

In this section we consider only special kinds of GPS, where the actions are of two types, **assert**(p) and **retract**(p), where p is an atom. The effect of **assert**(p) (resp. **retract**(p)) on a state S is adding (resp. removing) p to S (resp. from S). Moreover, the rules have the following structure:
$\neg p, l_1, \ldots, l_k, \textit{not } l_{k+1}, \ldots, \textit{not } l_n \rightarrow \textbf{assert}(p)$
$p, l_1, \ldots, l_k, \textit{not } l_{k+1}, \ldots, \textit{not } l_n \rightarrow \textbf{retract}(p)$
where l_i's are classical literals. Rules of the first kind are called *assert* rules, and rules of the second kind are called *retract* rules. If it is not important to distinguish between assert and retract rules, we will simply write a rule as
$l, l_1, \ldots, l_k, \textit{not } l_{k+1}, \ldots, \textit{not } l_n \rightarrow a$.
Taking inspiration from the notion of stratification in logic programming (Apt, Blair, & Walker 1988), we define *stratified* GPS's in such a way that negation as failure can be computed bottom-up. In the following, given a classical literal l, we refer to *the atom of* l as l if it is a positive atom, and as p if l is $\neg p$.

Definition 13 A GPS P is *stratified* if there exists a partition $P_0 \cup \ldots \cup P_n$ of its rules such that the following conditions are satisfied. Let
$l, l_1, \ldots, l_k, \textit{not } l_{k+1}, \ldots, \textit{not } l_h \rightarrow a$
be a rule in P_j. Then
(i) for each l_i, $i = 1, \ldots, k$, each rule containing the atom of l_i in the head must belong to $\bigcup_{m \leq j} P_m$
(ii) for each l_i, $i = k+1, \ldots, h$, each rule containing the atom of l_i in the head must belong to $\bigcup_{m < j} P_m$. ◇

For stratified GPS's, the grounded, preferred and stable semantics coincide (see theorem 15). Moreover, this semantics can be computed in a bottom-up way, by a simple operator $\mathcal{S}$, that we define next.

Let C be a possible partial computation
$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_n} S_n$.
Then for any h, k such that $0 \leq h < k \leq n$, the sequence
$S_h \xrightarrow{r_{h+1}} S_{h+1} \ldots \xrightarrow{r_k} S_k$
is called a subcomputation of C.

Definition 14 Let $P = P_0 \cup \ldots \cup P_n$ be a stratified GPS. Let $\mathcal{S}$ be the operator defined as follows.
$\mathcal{S}(P_0) = \mathcal{C}(P_0)$
$\mathcal{S}(P_0 \cup \ldots \cup P_{i+1}) = \{C \in \mathcal{C}(P_0 \cup \ldots \cup P_{i+1}) \mid$

- $*$ for each subcomputation C' of C if $C' \in \mathcal{C}(P_0 \cup \ldots \cup P_i)$ then $C' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$
- $*$ for each subcomputation of C of the form $S_{j-1} \xrightarrow{r_j} S_j$ such that $r_j \in P_{i+1}$ then for each *not* $l \in \textit{hyp}(r_j)$, there is no computation

$C' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ such that $initial(C') = S_{j-1}$ and $final(C') \models l\}$ ◇

Roughly speaking, the operator $\mathcal{S}$ formalizes the intuition that the acceptability of a possible partial computation using rules in $P_0 \cup \ldots \cup P_{i+1}$ depends only on computations in $P_0 \cup \ldots \cup P_i$. Thus, the semantics of a stratified GPS P can be computed bottom-up by iterating the operator $\mathcal{S}$ on the strata of P.

Theorem 15 Let P be a stratified GPS. Then:
(*i*) $\mathcal{S}(P)$ is grounded
(*ii*) $\mathcal{S}(P)$ is the unique preferred set of computations
(*iii*) $\mathcal{S}(P)$ is the unique stable set of computations ◇

Conclusions and future work

Production systems with negation as failure to find a course of actions are a natural extension of classical production systems, which increases their expressiveness in the sense that they allow a natural and simple representation (specification) of many real life problems. This extension can be given a simple semantics based on an argumentation theoretic framework. There are still several issues which deserve a deeper study and understanding.

We have seen that our semantics reflects the inherent nondeterminism of production systems. In fact, in our semantics different complete computations starting from the same initial state can yield different final states, even for stratified GPS's. This contrasts with many efforts in the literature aiming at finding a method to select one of the complete computations as the expected semantics (Froidevaux 1992; Raschid 1994). Even though we believe that in many cases these efforts contrast with the inherent nondeterministic nature of the problems represented by the production rules, there are situations in which selecting only one out of (possibly) many complete computations may not harm at all. In these cases, it is worth studying computational strategies which basically provide us with a deterministic operational semantics for production systems. Still, the declarative semantics serves as a basis for reasoning about the correctness of these methods.

We are investigating the application of our approach in the active databases area. Active databases (Ceri, Dayal, & Widom 1995) is an important research topic in the database community due to the fact that they find many applications in practice. Typical active database rule is an event-condition-action form. We are currently extending our argumentation based approach to active rules which also contain negation as failure in the condition part of the rules.

Finally, a few words about the relationship between "negation as failure to find a course of actions" presented in this paper and "negation as failure to prove" in logic programming. In (Dung 1995), it has been showed that every argumentation framework can be represented by a simple logic program using negation as failure to prove. This means that negation as failure to find a course of actions can be represented using naf to prove. In a following paper we have also showed that naf to prove is a special kind of negation as failure to find a course of actions. Hence the conclusion is that the two kinds of naf have the same expressive power. Which one should be used in a concrete application depends on which one allows a more natural specification of the problem at hand.

Acknowledgments This research was supported in part by EEC Keep in Touch Activity KIT011-LPKRR.

References

Apt, K.; Blair, H.; and Walker, A. 1988. Towards a theory of declarative knowledge. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann.

Bondarenko, A.; Dung, P.; Kowalski, R.; and Toni, F. 1995. An abstract, argumentation-based theoretic approach to default reasoning. Technical report, Dept. of Computing, Imperial College of Science, Technology and Medicine, London.

Ceri, S.; Dayal, U.; and Widom, J. 1995. *Active Database Systems.* Morgan-Kauffman.

Dung, P. 1995. The acceptability of arguments and its fundamental role in logic programming, nonmonotonic reasoning and n-person games. *Artificial Intelligence* 77(2).

Forgy, C. 1981. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University.

Froidevaux, C. 1992. Default logic for action rule-based systems. In Neumann, B., ed., *Proc. 10th European Conference on Artificial Intelligence (ECAI 92)*, 413–417. John Wiley & Sons.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Proceedings of the Fifth International Conference on Logic Programming*, 1070–1080. The MIT Press.

Hayes-Roth, F. 1985. Rule based systems. *Communications of the ACM* 28(9).

McDermott, D., and Doyle, J. 1980. Non-Monotonic Logic I. *Artificial Intelligence* 13(1-2):41–72.

Moore, R. 1985. Semantical considerations on nonmonotonic logics. *Artificial Intelligence* 25(1):75–94.

Raschid, L. 1994. A semantics for a class of stratified production system programs. *Journal of Logic programming.*

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* 13:81–132.

Van Gelder, A.; Ross, K.; and Schlipf, J. 1988. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh Symposium on Principles of Database Systems, ACM-SIGACT-SIGCOM*, 221–230. The MIT Press.