Production Systems with Negation As Failure

Phan Minh Dũng and Paolo Mancarella

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 non-monotonic 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 to find a course of actions can be computed by a simple bottom-up operator. Stratified production systems can be implemented on top of conventional production systems in two ways. The first way corresponds to the understanding of stratification as a form of priority assignment between rules. We show that this implementation, though sound, is not complete in the general case. Hence we propose a second implementation by means of an algorithm which transforms a finite stratified production system into a classical one. This is a sound and complete implementation, though computationally hard as shown in the paper.

Keywords: rule-based systems, knowledge-based systems, rule-based processing, expert systems, knowledge representation

Note: This paper is a revised and extended version of [9].

1 Introduction and Motivations

In this section we first give examples to motivate the extension of the production systems paradigm [17] 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.

1.1 On the need for negation as failure in production systems

Example 1.1
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.

Clearly, there are other ways of representing this situation which do not use negation as failure at all, as we will see in the next Section. We will argue, however, that the representation using negation as failure provides a better specification for the problem at hand. □

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

Example 1.2
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(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 common-sense 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 1.3
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. □

1.2 Negation as failure as a specification mechanism for reactive systems

Let us consider again the example 1.1. 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 *common-sense 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.

As we already pointed out, negation as failure can be seen as a mechanism to specify priorities between different goals. For example, in the robot firefighter example 1.3, negation as failure is used to give the goal of saving humans a higher priority than the goal of saving artifacts.

Explicit priorities between rules is often used in production systems and active database systems [2, 16, 11] to influence the way rules are executed. In such systems, whenever different rules can be triggered in a state, the rules which have higher priority are triggered first. Clearly, the notion of priority that negation as failure induces, is different from the one used in classical production and active database systems, as the former has to do with goals whether the latter has to do with rules which are employed to implement goals. Moreover, it is often difficult to understand declaratively why a rule should have higher priority than another rule. We believe that in many cases negation as failure can be used as a high-level tool to specify the (implicit) priority between goals which could then be implemented by defining explicit priorities between rules.

1.3 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, which is an extended and revised version of [9], 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 [8], which has been successfully adopted to understand logic programming with negation as failure as well as many other non-monotonic 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 1.1, 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 1.1, 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 [8]. This provides us with many natural semantics, such as the grounded (well-founded), the preferred and the stable semantics [8]. These semantics are arguably the most popular and widely accepted semantics for non-monotonic and common-sense reasoning in the literature [15, 4, 19, 27].

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.

We show that classical production systems with explicit priorities between rules can be used to obtain a straightforward *sound* implementation of stratified production systems with negation as failure. We will also show that a *complete* implementation requires a more sophisticated method of compiling away negation as failure even for the class if stratified production systems. This method yields a classical production systems, but its complexity, in the worst case, is not polynomial in the number of atoms occurring in the production system.

The rest of the paper is structured as follows. In Section 2 we introduce the basic notations and terminologies we use for classical production rule systems. In Section 3 we extend production systems to general production systems where negation as failure can be used in the condition part of the rules, and we provide them with an argumentation based semantics (argumentation systems are briefly reviewed in section 3.1). In Section 4 we address the problem of computing with negation as failure and introduce the class of stratified production systems, the semantics of which can be characterized by a simple bottom-up operator. In Section 5 we discuss two ways of implementing stratified production systems, along the lines mentioned above. Finally, in Section 6 we address some open issues and future work.

2 Preliminaries: Classical production rule systems

We introduce here the notations and basic terminologies we are going to use in the following. The production systems language we use is similar to classical ones (see, e.g. [12]). 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

$$\mathit{effect} : \mathcal{A} \times \mathit{Stat} \rightarrow \mathit{Stat}.$$

A *production rule* is a rule of the form

$$\textbf{if } l_1, \ldots, l_n \textbf{ then } a$$

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 $\mathit{cond}(r)$ (resp. $\mathit{action}(r)$). 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 2.1 (*Computations*)
A *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 $S_i = \mathit{effect}(\mathit{action}(r_i), S_{i-1})$. S_0 will be referred to as $\mathit{initial}(C)$ and S_n as $\mathit{final}(C)$. □

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

Definition 2.2 (*Complete Computations*)
A computation C

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

is called a *complete* computation if there is no production rule in P which is applicable in S_n. □

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 2.3 (*Input-Output semantics*)
For a production system P, the input-output semantics of P is defined by

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

Even though we have considered only ground production rules, our approach can be easily adopted for production systems in which rules may contain variables. Given such a collection of rules, we consider all their possible ground instances, obtained by replacing the variables occurring in them by any ground term. This approach is usually adopted in the study of the semantics of logic programming [15, 27, 7].

It is worth noting that the above semantics reflects the inherent nondeterministic nature of production systems. It is indeed natural to expect that from an initial state S there may exist many different computations leading to possibly different final states. This means that, for a given S, there might be many different pairs (S, S') in the $\mathcal{IO}$ semantics. This nondeterminism arises naturally in the representation of real life problems through production rules. Referring back to example 1.1 if both the shops are open and the neighbor is in there are two possible ways of acquiring machine powder, which are both plausible, i.e. buying it from the shop or borrowing it from the neighbor. Clearly the corresponding computations of the production system are both plausible and there is no intuitive reason to prefer one to the other.

The need for nondeterministic rule based languages has been pointed out by many authors (see, e.g. [1]). Indeed, nondeterministic rule based languages have been mainly studied with respect to their expressive power and computational complexity as opposed to purely deterministic languages. In this paper, we argue that nondeterminism is needed to naturally represent real life problems. Some work in the literature (see, e.g. [13, 24]) has also been devoted to define (operational) semantics for classical production systems, in such a way that nondeterminism is avoided by adopting an ad hoc computational mechanism which (either implicitly or explicitly) assigns a sort of priority to the production rules which can be fired in the same state. But the lack of an intuitive motivation makes a full understanding of their technical results difficult.

3 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 3.1 (*General production rules*)
A *general* production rule has the form

$$\textbf{if } l_1, \ldots, l_n \textbf{ then } a_i$$

where each l_i is a ground general literal. □

Given a general production rule

$$\textbf{if } l_1, \ldots, l_n \textbf{ then } a_i$$

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 *possibly applicable* in a state S if $S \models cl\text{-}cond(r)$[1].

Definition 3.2 (*Possible Computations*)
Given a GPS P, a *possible 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 $S_i = \mathit{effect}(\mathit{action}(r_i), S_{i-1})$.
We denote by $\mathcal{C}(P)$ the set of all possible computations of a GPS P. □

Given a possible computation as above, the sequence $\langle hyp(r_1), \ldots, hyp(r_n)\rangle$ is referred to as the sequence of hypotheses *underlying* the computation. 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 computation is then an acceptable 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 1.1.

[1]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.

Example 3.3
Let us first recall the production rules:

r_1: **if** *not* Machine-Powder **then** hand-wash
r_2: **if** Machine-Powder **then** machine-wash
r_3: **if** $\neg$ Machine-Powder, Shop-Open **then** buy-powder
r_4: **if** $\neg$ Machine-Powder, Neighbor-In **then** borrow-powder

The effects of the actions are specified below:

$$\mathit{effect}(\text{hand-wash}, S) = S \cup \{\text{Clothes-Clean}\}$$
$$\mathit{effect}(\text{machine-wash}, S) = \begin{cases} (S \cup \{\mathit{Clothes\text{-}Clean}\} & \text{if Machine-Powder} \in S \\ \\ S & \text{if Machine-Powder} \notin S \end{cases}$$
$$\mathit{effect}(\text{buy-powder}, S) = S \cup \{\text{Machine-Powder}\}$$
$$\mathit{effect}(\text{borrow-powder}, S) = S \cup \{\text{Machine-Powder}\}$$

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 computations starting from S_0, namely

$$C_1: \quad S_0 \xrightarrow{r_1} \{\text{Neighbor-In, Clothes-Clean}\}$$
$$C_2: \quad S_0 \xrightarrow{r_4} \{\text{Neighbor-In, Machine-Powder}\} \xrightarrow{r_2} \{\text{Neighbor-In, Machine-Powder, Clothes-Clean}\}.$$
$$C_3: \quad S_0 \xrightarrow{r_4} \{\text{Neighbor-In, Machine-Powder}\}$$

First, notice that both C_2 and C_3 are not based on any assumption. Our common-sense dictate that C_2 and C_3 represent acceptable course of actions from the initial state which lead to the common-sense 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* Machine-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* Machine-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 computations. In the following, we first recall the general notion of argumentation systems from [8] and then we show that the natural semantics of GPS can be defined using the theory of argumentation.

3.1 Argumentation systems

Argumentation has been recognized lately as an important and natural approach to nonmonotonic reasoning [5, 14, 18, 21, 22, 23, 26, 28]. It has been shown [8] that many major nonmonotonic logics [20, 19, 25] represent in fact different forms of a simple system of argumentation reasoning. Based on the results in [8], a simple logic-based argumentation system has been developed in [4] which captures well known nonmonotonic logics like autoepistemic logics, Reiter's default logics and logic programming as special cases. In [14], argumentation has been employed to give a proof procedure for conditional logics. Argumentation has also been applied to give an elegant semantics for reasoning with specificity in [10].

We review here the basic notions and definitions of argumentation systems (the reader can refer to [8] 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 $B \in H$.

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. 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 maximal conflict-free 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 non-monotonic 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.

In [8] it has been shown that logic programming with negation as failure can be seen as a form of argumentation systems. In this view, the various semantics of argumentation systems presented above capture in a unifying framework various well-known semantics for logic programming with negation as failure. For instance, the grounded semantics corresponds to the well-founded semantics in logic programming [27] and the stable semantics corresponds to stable semantics of logic programming [15].

In the following we show that general production systems with two kinds of negation are also a form of argumentation systems. A philosophical explanation of this result can be seen in the fact that the computations of production systems represent also a form of common-sense reasoning.

3.2 Computations as arguments

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

Definition 3.4 (*Attacks*)
Let C be a possible computation

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

An *attack* against C is a possible computation C' such that $\mathit{initial}(C') = S_i$, for some i, and there exists an underlying assumption *not* l in $\mathit{hyp}(r_{i+1})$ such that l holds in $\mathit{final}(C')$. □

Remark 3.5 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 lead by a computation to a state in which l holds. Referring back to example 3.3, it is easy to see that C_3 attacks C_1, by defeating the assumption "*not* Machine-Powder" on which C_1 is based. This is because such an assumption is made in the state S_0 and C_3 shows an alternative course of actions leading S_0 to a state where Machine-Powder actually holds. Consider now an initial state $S_0' = \emptyset$, representing, among others, the fact that there is no machine powder in house, shops are closed and the neighbor is not in. It is clear that there is only one possible computation (apart from the empty one) leading S_0' to the final state {Clothes-Clean}. This state represents the fact that clothes have been hand washed. This computation is also based on the assumption "*not* Machine-Powder" which is made in the state S_0' and hence cannot be defeated (the powder cannot be bought since shops are closed neither it can be borrowed since the neighbor is not in).

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. Recall that, given a set H of arguments (i.e. possible computations), $Def(H)$ is the set of all arguments which are defended by H (see Section 3.1).

Definition 3.6 Let P be a GPS and K be a set of possible computations. Then:

- K is grounded if K is the least solution of the equation $H = Def(H)$
- K is preferred if K is a maximal conflict-free solution of the equation $H = Def(H)$
- K is stable if K is conflict-free and it attacks every possible computation not in K.

□

Let us elaborate in detail the example 3.3 to compute its grounded semantics.

Example 3.7
We compute the grounded semantics by using the Def operator. We use the following abbreviations: MP for Machine-Powder, NI for Neighbor-In, CC for Clothes-Clean, SO for Shop-Open, hw for the action hand-wash, mw for the action machine-wash, bp for the action Buy-Powder and bop for the action borrow-powder. Hence the rules of example 3.3 become:

r_1: **if** *not* MP **then** hw
r_2: **if** MP **then** mw
r_3: **if** $\neg$ MP, SO **then** bp
r_4: **if** $\neg$ MP, NI **then** bop

As mentioned in the previous section, the grounded semantics is the least solution of the equation $H = Def(H)$. So, let H_0 be the empty set of arguments (computations). Clearly $H_1 = Def(H_0)$ contains all the computations which can never be attacked (H_0 cannot defend any computation). These are clearly all the empty computations and all the non-empty computations which are not based on any assumptions. Let S_{so} be any state such that $S_{so} \models \neg MP$ and $S_{so} \models SO$. Then the following computations

$$S_{so} \xrightarrow{r_3} S_{so} \cup \{MP\} \tag{a}$$

$$S_{so} \xrightarrow{r_3} S_{so} \cup \{MP\} \xrightarrow{r_2} S_{so} \cup \{CC\} \tag{b}$$

belong to H_1, since both $hyp(r_3)$ and $hyp(r_2)$ are empty. Similarly, let S_{ni} be any state such that $S_{ni} \models \neg MP$ and $S_{ni} \models NI$. Then the following computations

$$S_{ni} \xrightarrow{r_4} S_{ni} \cup \{MP\} \tag{c}$$

$$S_{ni} \xrightarrow{r_4} S_{ni} \cup \{MP\} \xrightarrow{r_2} S_{ni} \cup \{CC\} \tag{d}$$

belong to H_1. Let us move now to $H_2 = Def(H_1)$. Clearly $H_1 \subseteq H_2$. Let us consider now the possible computations with some underlying assumptions.

All such computations must use rule r_1, which is the only rule containing a naf-literal in its premises. Let S_1 be any state such that $S_1 \models \neg MP$. Consider then a computation C of the form

$$S_1 \xrightarrow{r_1} S_1 \cup \{CC\} \tag{e}$$

which is based on the assumption *not* MP. Now, if $S_1 \models SO$, then there is a computation of the form (a) above, which attacks this computation (since it leads to a final state where MP holds) and belongs to H_1. Since H_1 is conflict-free, H_1 cannot defend the computation C. Similarly, if $S_1 \models NI$, then H_1 cannot defend C. So, we conclude that the only computations in $H_2 \setminus H_1$ are the computations of the form (e) such that $S_1 \models \neg SO, \neg NI$.

It is easy to see that H_2 is indeed a solution, hence the least solution, of the equation $H = Def(H)$ in this case. Moreover it is not difficult to see that, in this example, the grounded, preferred and stable semantics coincide, and hence H_2 is also stable and preferred. This is also a consequence of the fact that this production system is stratified in the sense of section 4, and we will show that for stratified production systems all different semantics coincide. □

It is easy to see that the following propositions hold.

Proposition 3.8
If a computation C attacks a computation C' and C' is a prefix of C'', then C also attacks C''. □

Proposition 3.9
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 give now some examples of general production systems, where the grounded, preferred and stable semantics do not coincide.

Example 3.10
Let P_0 be the following GPS:

$$\begin{array}{ll} r_1: & \neg b, not\ a \rightarrow \mathbf{assert}(b) \\ r_2: & \neg a, not\ b \rightarrow \mathbf{assert}(a) \end{array}$$

where the effect of $\mathbf{assert}(p)$ on a state S is adding p to S. Let $S_0 = \emptyset$. It is easy to see that the nonempty possible computations starting from S_0 are:

$$\begin{array}{ll} C_1 : S_0 \xrightarrow{r_1} \{b\} & \text{with underlying assumptions } \langle\{not\ a\}\rangle \\ C_1' : S_0 \xrightarrow{r_1} \{b\} \xrightarrow{r_2} \{a,b\} & \text{with underlying assumptions } \langle\{not\ a\},\{not\ b\}\rangle \\ C_2 : S_0 \xrightarrow{r_2} \{a\} & \text{with underlying assumptions } \langle\{not\ b\}\rangle \\ C_2' : S_0 \xrightarrow{r_2} \{a\} \xrightarrow{r_1} \{a,b\} & \text{with underlying assumptions } \langle\{not\ b\},\{not\ a\}\rangle \end{array}$$

The empty computation starting from $\{b\}$ attacks C_1' and the empty computation starting from $\{a\}$ attacks C_2'. On the other hand, an empty computation cannot be attacked. Hence, C_1' and C_2' are not contained in any acceptable set of computations. Furthermore, it is also clear that C_2 attacks C_1, since C_1 is based on the assumption *not* a, meaning that there is no way to achieve a starting from S_0. For similar reasons, C_1 also attacks C_2. Hence, the only computation starting from S_0 which is contained in the grounded semantics is the empty computation.

But there are two stable sets of computations, one containing C_2 and the other containing C_1. This is because C_1 (resp. C_2) can defend itself against all attacks. In this example preferred and stable semantics coincide.

Notice that rules similar to r_1 and r_2 may be needed to represent real world situations. Imagine the situation of a team leader who needs to hire a person for an important position and he has two applications for it, say *Mary* and *Ann*. The leader asks his advisors to express their opinion. The first advisor likes *Ann*, hence he says: "if there is no way you can hire Ann, then hire Mary". This can be expressed by the rule

$Advisor_1$: **if** ¬ Mary_hired, *not* Ann_hired **then** Hire(Mary)

Notice that using negation as failure we capture here the intuition that there is no possible course of actions to hire Ann. Imagine now that the second advisor has exactly the opposite view, i.e. his opinion can be expressed as:

$Advisor_2$: **if** ¬ Ann_hired, *not* Mary_hired **then** Hire(Ann)

The team leader reasoning is represented by two computations which corresponds to computations C_1 and C_2 above. □

The next example shows that preferred sets may not be stable.

Example 3.11
Let P_1 be the following general production system.

$$r: \quad \neg a, not\ a \rightarrow \mathbf{assert}(a).$$

The intuition of this rule is that if a is not true in the current state and there is no way to achieve a, then add it to the state. This is clearly a paradox, since if there is no way to achieve a then the rule itself allows to achieve it. The only nonempty possible computation starting from $S_0 = \emptyset$ is

$$C : S_0 \xrightarrow{r} \{a\}.$$

It is clear that C attacks itself. Hence, in this example the grounded and the preferred semantics coincide, and contain only the empty computations. Clearly, there is no stable set of computations. □

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

Definition 3.12 (*Complete Computations*)
Let P be a GPS and *Sem* be a selected semantics of P, i.e. *Sem* is either the grounded, or a preferred, or a stable, set of possible computations. A computation $C \in Sem$ is called a *Sem-complete* computation if there exists no other computation $C' \in Sem$ such that C is a prefix of C'. □

If all the semantics of a GPS coincide, we simply talk about *complete* computations instead of *Sem*-complete computations. Referring back to the example 3.3, 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 3.13 (*Input-Output semantics for GPS*)
Let P be a GPS.

- The grounded input-output semantics of P is defined by
  $\mathcal{IO}_G(P) = \{(\mathit{initial}(C), \mathit{final}(C)) \mid C \text{ is a grounded complete computation}\}$
- Let *Sem* be a set of arguments which is preferred or stable. Then
  $\mathcal{IO}_{Sem}(P) = \{(\mathit{initial}(C), \mathit{final}(C)) \mid C$ is a *Sem*-complete computation of P $\}$

$\square$

4 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:

$$\neg p, l_1, \ldots, l_k, \mathit{not}\ l_{k+1}, \ldots, \mathit{not}\ l_n \rightarrow \mathbf{assert}(p)$$

$$p, l_1, \ldots, l_k, \mathit{not}\ l_{k+1}, \ldots, \mathit{not}\ l_n \rightarrow \mathbf{retract}(p)$$

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

$$l, l_1, \ldots, l_k, \mathit{not}\ l_{k+1}, \ldots, \mathit{not}\ l_n \rightarrow a.$$

Further, 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 4.1 (*Stratified Production Systems*)
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

$$l, l_1, \ldots, l_k, \mathit{not}\ l_{k+1}, \ldots, \mathit{not}\ l_h \rightarrow a$$

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 $\bigcup\limits_{m<j} P_m$.

□

It is worth noting that the intuition underlying the above definition of stratified GPS's is similar to the one underlying the definition of stratification in logic programming [3].

For stratified GPS's, the grounded, preferred and stable semantics coincide (see theorem 4.3). Moreover, this semantics can be computed in a bottom-up way, by a simple operator $\mathcal{S}$, that we are going to define next.

Let C be a possible 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. Further, we denote by $rules(C)$ the sequence $r_1, \ldots, r_n$.

Definition 4.2 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 $C' \in \mathcal{C}(P_0 \cup \ldots \cup P_i)$, $C' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$
- 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* $l \in hyp(r_j)$, 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 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 4.3
Let P be a stratified GPS. Then:

- $\mathcal{S}(P)$ is grounded
- $\mathcal{S}(P)$ is the unique preferred set of computations
- $\mathcal{S}(P)$ is the unique stable set of computations

□

Proof. : See appendix.

5 Implementing Stratified PS

In this section we address the issue of implementing stratified PS by translating stratified PS into classical PS. From now on, we restrict ourselves to production systems where the collection of (the ground instances) of rules is finite.

Let us first introduce the notion of *implementation* of a production system. Let P and P' be two production systems. We say that P' is a *sound* implementation of P if the following two conditions hold:

(i) for each computation C' of P' there exists a computation C of P such that $\mathit{initial}(C') = \mathit{initial}(C)$ and $\mathit{final}(C') = \mathit{final}(C)$.

(ii) for each complete computation C' of P' there exists a complete computation C of P such that $\mathit{initial}(C') = \mathit{initial}(C)$ and $\mathit{final}(C') = \mathit{final}(C)$.

If P' is a *sound* implementation of P, we say it is also *complete* if the reverse of conditions (i) and (ii) above hold, namely:

- for each computation C of P there exists a computation C' of P' such that $\mathit{initial}(C) = \mathit{initial}(C')$ and $\mathit{final}(C) = \mathit{final}(C')$.
- for each complete computation C of P there exists a complete computation C' of P' such that $\mathit{initial}(C) = \mathit{initial}(C')$ and $\mathit{final}(C) = \mathit{final}(C')$.

In the next section we introduce the class of *prioritized* production systems PPS where rules can have different priorities, and define accordingly a suitable notion of computation. We show that viewing stratification as a priority assignment to rules (the lower the stratum, the higher the priority), yields a sound implementation of a stratified GPS through PPS. However, we will see by means of a simple example that this implementations is not complete in the general case. This means that the priorities induced by stratification are not powerful enough to completely capture the implicit priorities induced by the negation as failure mechanism. In order to obtain a sound and complete implementation, we need a more sophisticated method which compiles away negation as failure from a stratified GPS and yields a classical production system with no priorities at all. We show that the classical PS obtained by this transformation is indeed a sound and complete implementation of the original stratified PS. However, the transformation, in the worst case, is not polynomial in the number of atoms of the production system.

5.1 Implementing negation as failure with priorities

Let us introduce the class of PPS, namely classical production systems where rules can be assigned different priorities. This type of systems have been extensively studied in the literature [11].

Definition 5.1 A *prioritized production system (PPS)* is a pair $\langle P, \prec \rangle$ where P is a classical PS and $\prec$ is a partial order relation between rules [2]. $\square$

[2] a partial order is a irreflexive, transitive and asymmetric relation

In the sequel, if $r \prec r'$, we will say that r has *higher* priority than r'. Notice that a classical PS is a PPS with $\prec$ being the empty relation.

The priority relation between rules affects the applicability of rules.

Definition 5.2 Let $\langle P, \prec \rangle$ be a PPS. A production rule **if** $l_1, \ldots, l_n$ **then** a is *applicable* in a state S iff $S \models l_1 \wedge \ldots \wedge l_n$ and there is no rule $r' \prec r$ such that r' is applicable in S. □

The notion of computations in PPS is the same as the one given in Definition 2.1, provided that applicability of rules is understood as in the previous definition.

Example 5.3 Let us formulate the washing machine example 3.3 as a PPS. Let us first recall the production rules:

r_1: **if** $\neg$ Machine-Powder **then** hand-wash
r_2: **if** Machine-Powder **then** machine-wash
r_3: **if** $\neg$ Machine-Powder, Shop-Open **then** buy-powder
r_4: **if** $\neg$ Machine-Powder, Neighbor-In **then** borrow-powder

Notice that negation as failure in the first rule has been replaced by classical negation. To correctly represent the fact that hand washing has lower priority than machine washing we can add the following priority relation between rules.

$$r_3 \prec r_1 \qquad\qquad r_3 \prec r_1$$

In this way, rule r_1 cannot be applied in states where there is no machine powder but either the shops are open or the neighbor is in, thus achieving the desired behavior. □

Let now P be a stratified GPS. The idea is to view stratification as an implicit assignment of priorities between rules: making this assignment explicit, allows us to get rid of negation as failure and translate it directly into classical negation. We need first to introduce some useful notations.

Let l be a naf literal. Then we denote by l^* the classical literal

- $\neg a$, if $l = not\ a$
- a, if $l = not\ \neg a$

Moreover, given a general production rule r, we denote by r^* the rule obtained from it by replacing each naf-literal l by l^*.

Definition 5.4 Let $P = P_0 \cup P_1 \cup \ldots \cup P_n$ be a stratified GPS. Its prioritized form $\langle P^*, \prec \rangle$ is the PPS obtained by replacing each rule r of P by r^* and by defining $\prec$ as follows:

$$r^* \prec s^* \text{ iff } r \in P_i \wedge s \in P_j \wedge i < j.$$ □

The following theorem shows that, given a stratified GPS P, $\langle P^*, \prec \rangle$ is a sound implementation of it.

Theorem 5.5 Let P be a stratified GPS and $\langle P^*, \prec \rangle$ its prioritized form. Then $\langle P^*, \prec \rangle$ is a sound implementation of P, i.e.

(i) for each computation C in $\langle P^*, \prec \rangle$ there exists a computation C' in P such that $initial(C) = initial(C')$ and $final(C) = final(C')$

(ii) for each complete computation C in $\langle P^*, \prec \rangle$ there exists a complete computation C' in P such that $initial(C) = initial(C')$ and $final(C) = final(C')$.

$\square$

Proof. : See appendix.

The following example shows that the implementation we have just given is not complete in the general case.

Example 5.6 Consider the following GPS P:

$r_1 : \quad \neg a, \neg d, not\ b \rightarrow \mathbf{assert}(a)$
$r_2 : \quad \neg b, \neg c, d \rightarrow \mathbf{assert}(b)$
$r_3 : \quad \neg c \rightarrow \mathbf{assert}(c)$
$r_4 : \quad \neg d \rightarrow \mathbf{assert}(d)$

and its stratification $P_0 \cup P_1$ where $P_0 = \{r_2, r_3, r_4\}$ and $P_1 = \{r_1\}$.

Consider the following complete computation

$\{\ \} \xrightarrow{r_3} \{c\} \xrightarrow{r_1} \{a, c\} \xrightarrow{r_4} \{a, c, d\}$

The prioritized form of P is given by the following PPS $\langle P^*, \prec \rangle$:

$r_1^* : \quad \neg a, \neg d, \neg b \rightarrow \mathbf{assert}(a)$
$r_2^* : \quad \neg b, \neg c, d \rightarrow \mathbf{assert}(b)$
$r_3^* : \quad \neg c \rightarrow \mathbf{assert}(c)$
$r_4^* : \quad \neg d \rightarrow \mathbf{assert}(d)$

where $\prec$ is defined by $r_i^* \prec r_1^*$, for each $i = 2, 3, 4$. It is not difficult to see that there is no complete computation in $\langle P^*, \prec \rangle$ starting from the empty state and ending in the state $\{a, c, d\}$. This is due to the fact that r_3^* and r_4^* are both rule with highest priority and then they are fired before r_1^*. Since the effect of r_4^* is adding $\{d\}$ to the state, this will prevent r_1^* from being applicable.

The point here is that the naf literal *not* b in r_1 induces a priority between rules which is different from the naive one obtained by the given stratification. Indeed, *not* b intuitively means that there is no way to achieve b and, given a state where all classical conditions of r_1 are satisfied, this can be ensured by enforcing c to be true, i.e. by making r_2, the only rule for asserting b not applicable. This can be obtained in two ways: either by enforcing the priority $r_3^* \prec r_1^*$ or by transforming r_1 into the following classical rule

$r_1' : \quad \neg a, \neg d, \neg b, \mathbf{c} \rightarrow \mathbf{assert}(a)$ $\square$

We thus need a deeper understanding of stratification, generalizing the intuition sketched in the above example which allowed us to compile away negation as failure in rule r_1'. In the next section we propose such a transformation, which, given a stratified GPS P, yields a classical PS P' which is a sound and complete implementation of P. However, in order to get a complete implementation we have to pay a high price in terms of the complexity of the transformation, as shown in Section 5.3.

5.2 Compiling away negation as failure

Let us first introduce the notion of incomplete state and computations with respect to incomplete states.

Definition 5.7 An *incomplete state* is a consistent set of ground literals of $\mathcal{L}$. A production rule **if** $l_1, \ldots, l_n$ **then** a is said to be *applicable* in an incomplete state I iff the conditions $l_1, \ldots, l_n$ are true in I, i.e. $I \models l_1 \wedge \ldots \wedge l_n$. □

Often it is not necessary to have complete information about the state of the world to carry out some computations. Indeed, if a computation relies only on classical production rules, the following notion of computation, which we refer to as *partial computation*, is sufficient.

Definition 5.8 (*Partial Computation*)
Let $I_0, \ldots, I_n$ $(n \geq 0)$ be incomplete states, and r_i's be production rules in a classical production system P. Then a sequence
$I_0 \xrightarrow{r_1} I_1 \ldots \xrightarrow{r_n} I_n$
is a *partial computation* C of P if for each $i \geq 1$, r_i is applicable in I_{i-1}, and

$$I_i = \begin{cases} I_{i-1} \cup \{p\} \setminus \{\neg p\} & \text{if } action(r_{i-1}) = \mathbf{assert}(p) \\ \\ I_{i-1} \cup \{\neg p\} \setminus \{p\} & \text{if } action(r_{i-1}) = \mathbf{retract}(p) \end{cases}$$

□

Notice that a computation is also a partial computation. Moreover, any partial computation C starting from an incomplete state I can be viewed as a *collection* of all computations C' starting from a state S such that $S \models I$ and $rules(C) = rules(C')$.

Definition 5.9 Let P be a classical PS, I be an incomplete state and l be a ground literal. Then we say that l is *achievable from* I in P if there is a partial computation $\mathcal{C}$ starting from I such that $l \in final(\mathcal{C})$. □

Let us first consider the problem of compiling away negation as failure for a stratified PS with only two strata $P = P_0 \cup P_1$.

Let r be a rule in P_1 of the following form

$$l_1', \ldots, l_k', not\ l_0, \ldots, not\ l_m, \rightarrow a$$

Let $I_1, \ldots, I_n$ be the collection of all incomplete states such that, for each $1 \leq i \leq n$, the following conditions are satisfied:

- $cl\text{-}cond(r) \cup I_i$ is consistent,

- l_0 is achievable from I_i in P_0

For each i, $1 \leq i \leq n$, let $J_i = I_i \setminus cl\text{-}cond(r)$. There are two cases here:

- some J_i is empty. This means that there exists a (classical) computation in P_0 starting from $cl\text{-}cond(r)$ and leading to a state in which l_0 holds. Hence, the rule r can never be applied in any computation in $\mathcal{S}(P)$ and so it can be simply dropped;

- no J_i is empty.

Let $\mathcal{R}_{l_0,r}$ be the empty set if one of the J_i is empty. Otherwise, $\mathcal{R}_{l_0,r}$ is the following set:

$$\mathcal{R}_{l_0,r} = \{R \mid \begin{array}{l} R \models \neg J_i, \text{ for each } i = 1, \ldots, n, \text{ and} \\ R \text{ is minimal}\} \end{array}$$

Notice that each non-empty J_i is viewed as a conjunction of literals. Also, by minimality, we mean that there exists no proper subset of R which implies $\neg J_i$ for each $i = 1, \ldots, n$.

Each set $R \in \mathcal{R}_{l_0,r}$ is a set of conditions that, if satisfied by a state, make l_0 not achievable from this state, and the set $\mathcal{R}_{l_0,r}$ covers all such conditions. This is formalized in the following lemma.

Lemma 5.10
Let S be a state such that $S \models cl\text{-}cond(r)$. Then, $S \models R$ for some $R \in \mathcal{R}_{l_0,r}$ if and only if there exists no computation C in $\mathcal{C}(P_0)$ such that $initial(C) = S$ and $final(C) \models l_0$. $\square$

Proof. From the definition of $\mathcal{R}_{l_0,r}$ it is clear that:
$S \models R$, for some $R \in \mathcal{R}_{l_0,r}$
if and only if
for all I such that l_0 is achievable from I and I is consistent with $cl\text{-}cond(r)$, $S \models \neg(I \setminus cl\text{-}cond(r))$.

It is clear now that if the above condition is satisfied, then there is no computation C in $\mathcal{C}(P_0)$ such that $initial(C) = S$ and $final(C) \models l_0$.

Suppose now that there is no computation C in $\mathcal{C}(P_0)$ such that $initial(C) = S$ and $final(C) \models l_0$. Assume now that there exists an incomplete state I such that l_0 is achievable from I, I is consistent with $cl\text{-}cond(r)$ and $S \not\models \neg(I \setminus cl\text{-}cond(r))$. From the assumption that $S \models cl\text{-}cond(r)$, it follows that $S \models I$, and hence there exists a computation C in $\mathcal{C}(P_0)$ such that $initial(C) = S$ and $final(C) \models l_0$. Contradiction. $\square$

For each $R \in \mathcal{R}_{l_0,r}$, we define a new production rule r_R as follows:

- $cl\text{-}cond(r_R) = cl\text{-}cond(r) \cup R$

- $hyp(r_R) = hyp(r) \setminus \{not\ l_0\}$

- $action(r_R) = action(r)$

We can now define:

$$P_{l_0,r} = \{r_R \mid R \in \mathcal{R}_{l_0,r}\}.$$

Example 5.11
The machine wash example 3.7 is translated into the following stratified PS.

r_1: **if** $\neg$ *CC*, *not MP* **then assert**(*CC*)
r_2: **if** $\neg$ *CC*, *MP* **then assert**(*CC*)
r_3: **if** $\neg$ *MP*, *SO* **then assert**(*MP*)
r_4: **if** $\neg$ *MP*, *NI* **then assert**(*MP*)

where $P = P_0 \cup P_1$, $P_0 = \{r_2, r_3, r_4\}$ and $P_1 = \{r_1\}$. Then, the set of all incomplete states from which *MP* is achievable is:

$$\{X \mid \{MP\} \subseteq X, \text{ or } \{\neg MP, SO\} \subseteq X, \text{ or } \{\neg MP, NI\} \subseteq X\}.$$

Hence,

$$\mathcal{R}_{MP,r_1} = \{\{\neg MP, \neg SO, \neg NI\}\}.$$

Hence, P_{MP,r_1} consists only of the following rule:

if $\neg$ *CC*, $\neg$ *MP*, $\neg$ *SO*, $\neg$ *NI* **then assert**(*CC*)

□

The production system $Q = P_0 \cup P_{l_0,r}$ is clearly still stratified, and it is equivalent to $P_0 \cup P_1$ as shown by the following results.

Lemma 5.12
$S \xrightarrow{r} S' \in \mathcal{S}(P)$ if and only if there exists $r' \in P_{l_0,r}$ such that
$S \xrightarrow{r'} S' \in \mathcal{S}(Q)$. □

Proof. Let us consider the following assertion:

(*) there is no computation $C \in \mathcal{S}(P_0)$ such that
$initial(C) = S$ and $final(C) \models l_0$.

(*) holds
if and only if
for each consistent set I of literals such that l_0 is achievable from I in P_0, $S \models \neg I$
if and only if (from Lemma 5.10)
there exists $R \in \mathcal{R}_{l_0,r}$ such that $S \models R$.

By definition of the operator $\mathcal{S}$:
$S \xrightarrow{r} S' \in \mathcal{S}(P)$
if and only if
$\forall l_i : i = 0, \ldots, m$ there is no computation $C \in \mathcal{S}(P_0)$ such that
$initial(C) = S$ and $final(C) \models l_i$

if and only if
  (*) and $\forall l_i : i = 1, \ldots, m$ there is no computation $C \in \mathcal{S}(P_0)$
  such that $initial(C) = S$ and $final(C) \models l_i$
if and only if
  there exists $R \in \mathcal{R}_{l_0,r}$ such that $S \models R$ and
  $\forall l_i : i = 1, \ldots, m$ there is no computation $C \in \mathcal{S}(P_0)$
  such that $initial(C) = S$ and $final(C) \models l_i$
if and only if
  $S \xrightarrow{r_R} S' \in \mathcal{S}(Q)$. □

Theorem 5.13
$S_0 \xrightarrow{r_0} S_1 \ldots S_{h-1} \xrightarrow{r_h} S_h \in \mathcal{S}(P)$ if and only if $S_0 \xrightarrow{r'_0} S_1 \ldots S_{h-1} \xrightarrow{r'_h} S_h \in \mathcal{S}(Q)$. □

Proof. Obvious, by taking $r'_j = r_j$ if $r_j \neq r$ and $r'_j \in \mathcal{R}_{l_0,r}$ as in the above lemma if $r_j = r$. □

The procedure of compiling away negation as failure from a two strata production system $P = P_0 \cup P_1$ is then the following:

Step 1. Select $r \in P_1$ and $not\ l \in hyp(r)$. If no such r exists then stop.
Step 2. if $hyp(r) = \{not\ l\}$ and $P_{l,r} \neq \emptyset$
    then $P_0 := P_0 \cup P_{l,r}$;
        $P_1 := P_1 \setminus \{r\}$
    else $P_1 := (P_1 \setminus \{r\}) \cup P_{l,r}$
Step 3. Goto Step 1.

The generalization of the above procedure for the general case of stratified production systems $P = P_0 \cup P_1 \cup \ldots \cup P_n$ with more than two strata is obvious. We first compile away naf from P_1 and obtain a stratified production system with $n-1$ strata. By continuing this process, we eventually obtain a classical production system.

Coming back to Example 5.11, the result of applying the above transformation is the classical PS consisting of the rules in the original P_0
  r_2: **if** $\neg$ *CC*, *MP* **then assert**(*CC*)
  r_3: **if** $\neg$ *MP*, *SO* **then assert**(*MP*)
  r_4: **if** $\neg$ *MP*, *NI* **then assert**(*MP*)

together with the new rule

  if $\neg$ *CC*, $\neg$ *MP*, $\neg$ *SO*, $\neg$ *NI*, **then assert**(*CC*).

It should be obvious that the new production system is equivalent to the original one.

In the above procedure, we assume that $P_{l,r}$ is given. We define now a method to compute it.

Definition 5.14 A set $\{I_1, \ldots, I_n\}$ of incomplete states is called a *base* for l with respect to P_0 if the following conditions hold:

- l is achievable from each I_i in P_0;
- for any incomplete state I such that l is achievable from I in P_0, $I \supseteq I_i$, for some $i = 1, \ldots, n$.

$\square$

For each $r \in P_1$ and $l \in hyp(r)$, it is easy to see that the following lemma holds.

Lemma 5.15
Let $\mathcal{I} = I_1, \ldots, I_n$ be a base for l with respect to P_0, and $J_i = I_i \setminus cl\text{-}cond(r)$, for each $i = 1, \ldots, n$. Suppose that no J_i is empty. Then

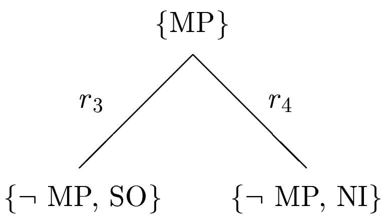
$$\mathcal{R}_{l,r} = \{R \mid \; R \models \neg J_i, \text{ for each } i = 1, \ldots, n, \text{ and } R \text{ is minimal}\}$$ $\square$

Constructing $P_{l,r}$ consists of computing a base for l with respect to P_0 and computing $\mathcal{R}_{l,r}$. The set $\mathcal{R}_{l,r}$ can be computed from a base by applying standard methods in propositional logic for computing disjunctive normal forms. In the following we give an algorithm for computing a base for l. We do so by constructing a tree satisfying the following properties:

- Each node N of the tree is labeled with an incomplete state $label(N)$.
- The root of the tree is labeled with the set $\{l\}$.
- Each link is labeled by a rule $r' \in P_0$.
- For any node M, a node N is a child of M with a link labeled by a rule $r' \in P_0$ iff M has no ancestor with the same label, $label(N)$ is consistent, and
  - if $action(r') = \textbf{assert}(p)$ then $p \in label(M)$ and $label(N) = cond(r') \cup label(M) \setminus \{p\}$;
  - if $action(r) = \textbf{retract}(p)$ then $\neg p \in label(M)$ and $label(N) = cond(r') \cup label(M) \setminus \{\neg p\}$.

It is clear from the above construction that if a node N has some ancestor with the same label then N is a leaf node. Hence, since P_0 is finite, the tree is also finite. Further, it should be clear from the construction of the tree that for each pair (P_0, l), there is exactly one tree satisfying the above conditions. Let us refer to this tree as $T_{P_0,l}$.

As an example, consider the production system of Example 5.11. Then $T_{P_0,\mathrm{MP}}$ is the following tree.



Theorem 5.16
$\{label(N) \,|\, \text{N is a node in } T_{P_0,l}\}$ is a base for l with respect to P_0. □

Proof. See appendix.

5.3 Complexity of the transformation

In this section we show that compiling away NAF is, in general, not polynomial in the number of atoms of the production system. The method we have given to compile away a naf literal *not* l occurring in a rule r basically consists of computing a base for l and then computing the set $\mathcal{R}_{l,r}$. The set $\mathcal{R}_{l,r}$ can be computed from a base by applying standard methods in propositional logic for computing disjunctive normal forms. We show next that computing the set $\mathcal{R}_{l,r}$ is not polynomial by taking into account a restricted class of stratified production systems.
First of all we will consider in the sequel classical production systems where each rule has the following structure:
$\neg b, a_1, \ldots, a_n \rightarrow \mathbf{assert}(b)$ (*)
where $b, a_1, \ldots, a_n$ are all positive atoms. In the sequel a rule of this form is referred to as a *rule for* b. Given two atoms a, b we say that a is a successor of b, denoted by $a < b$ if a occurs in the body of a rule for b. A production system is *acyclic* if the relation $<$ is well founded, i.e. each decreasing sequence of the form

$$\ldots a_n < a_{n-1} < \ldots < a_1 < a_0$$

is finite.

Give an acyclic production system, we define the *rank* of an atom a, denoted by $\|a\|$, as follows:

(1) $\|a\| = 0$ if $\{b \,|b < a\} = \emptyset$.

(2) $\|a\| = 1 + max\{\|b\| \;\; | \; b < a\}$, otherwise.

Let now $P = P_1 \cup P_0$ be a stratified production system where:

- P_1 contains exactly one rule
  r: $\neg p, not \;\; l \rightarrow \mathbf{assert}(p)$
- P_0 is a classical production system satisfying the following conditions:
  (c1) P_0 is acyclic

(c2) each rule in P_0 has the following structure:

$\neg a, a_1, a_2 \rightarrow$ **assert**(a)

(c3) each atom either appears in the head of exactly two rules or it appears in the head of no rule

(c4) a positive atom either occurs in the body of exactly one rule or it does not occur in the body of any rule

(c5) for each atom a, either $\|a\| = 0$ or $\|a\| = \|b\| + 1$, for each $b < a$.

(c6) for each atom a there exists a decreasing sequence
$a = a_0 < a_1 < \ldots < a_n = l$
where l is the atom occurring in the naf literal of rule r .

Notice that condition (c5) implies that for any atoms a, b, c, if both $b < a$ and $c < a$ then $\|b\| = \|c\|$. Notice also that each atom has either no successors or exactly four successors due to conditions (c2) and (c3) above.
We want to show that the number of rules obtained by compiling away the naf literal from rule r is not polynomial in the number of atoms of P_0. This amounts at proving that the cardinality of the set $\mathcal{R}_{l,r}$ is not polynomial in the number of atoms of P_0. Recall that the set $\mathcal{R}_{l,r}$ can be computed from a base for l by applying standard methods in propositional logic for computing disjunctive normal forms.

Let a be an atom and Γ be a partial state satisfying the following conditions:

(i) a is not achievable from any state satisfying Γ

(ii) Γ is minimal, i.e. no proper subset of it satisfies (i)

Let also $\mathcal{R}_a$ be the set of partial states satisfying conditions (i) and (ii) above. Then obviously $\mathcal{R}_{l,r} = \mathcal{R}_l$. In the sequel we show that the cardinality of $\mathcal{R}_l$ is not polynomial in the number of atoms of P_0.

First of all, let us generalize the above definition of $\mathcal{R}_l$ to a conjunction of atoms. Given a conjunction $a_1 \wedge \ldots \wedge a_n$, $\mathcal{R}_{a_1,\ldots,a_n}$ is the set of all partial states Γ satisfying the following conditions:

(iii) $a_1 \wedge \ldots \wedge a_n$ is not achievable from any state satisfying Γ

(iv) Γ is minimal, i.e. no proper subset of it satisfies (iii)

Consider now a conjunction of distinct atoms $a_1, \ldots, a_n$ such that $\|a_1\| = \ldots = \|a_n\|$ and let Γ be a partial state in $\mathcal{R}_{a_1,\ldots,a_n}$. By the syntactic restrictions on P_0 it is clear that $\Gamma \in \mathcal{R}_{a_i}$ for some a_i. Moreover it is clear that $\mathcal{R}_{a_i}$ and $\mathcal{R}_{a_j}$, with $i \neq j$, are disjoint sets.
For $i = 1, \ldots, n$, let m_i be the cardinality of $\mathcal{R}_{a_i}$. By the previous observations, it is clear that the cardinality of $\mathcal{R}_{a_1,\ldots,a_n}$ is $m_1 + m_2 \ldots + m_n$.

Consider now an atom b such that $\|b\| > 0$. Then P_0 contains exactly two rules for b,

$\neg b, a_1, a_2 \rightarrow \textbf{assert}(b)$
$\neg b, a_3, a_4 \rightarrow \textbf{assert}(b)$

Consider a partial state $\Gamma \in \mathcal{R}_b$. By the syntactic restrictions on P_0 and the minimality of Γ, it is clear that Γ can be split into two disjoint sets Γ' and Γ'' such that $\Gamma' \in \mathcal{R}_{a_1,a_2}$ and $\Gamma'' \in \mathcal{R}_{a_3,a_4}$.

Recall that, if both $a < b$ and $c < b$, then $\|a\| = \|c\| = \|b\| - 1$. Let now P_{a_i}, for $i = 1, 2, 3, 4$, be the set of rules of P_0 which are rules for a_i or rules for any atom c such that $c < \ldots\ldots < a_i$. By the structure of P_0 and the fact that $\|a_1\| = \|a_2\| = \|a_3\| = \|a_4\| = \|b\| - 1$, it is clear that P_{a_i} and P_{a_j}, $i \neq j$ are equal up to the renaming of atoms. Hence it is also clear that the sets $\mathcal{R}_{a_i}$, $\mathcal{R}_{a_j}$, $i \neq j$ are equal up to the renaming of atoms. Let then m be the cardinality of $\mathcal{R}_{a_i}$, $i = 1, 2, 3, 4$. By the previous arguments, it is obvious that the cardinality of both $\mathcal{R}_{a_1,a_2}$ and $\mathcal{R}_{a_3,a_4}$ are $2 \cdot m$. Hence, the cardinality of $\mathcal{R}_b = (2 \cdot m)^2$.

Let now $k = \|b\|$. We have just seen that the cardinality of $\mathcal{R}_b$ can be given inductively as

$f(k) = (2 \cdot f(k-1))^2$, for $k > 0$.

Furthermore it is obvious that $f(0) = 1$, since any atom a such that $\|a\| = 0$ does not occur in the head of any rule and hence $\{\neg a\}$ is the only set in $\mathcal{R}_a$.

It is easy to show by induction that the relation $f(k)$ we have just defined has the unique solution

$$f(k) = 4^{2^k - 1}$$

Let us now go back to the rule of P_1

r: $\neg p, \mathit{not}\ l \rightarrow \textbf{assert}(p)$.

Let $\|l\| = k$. It is not difficult to see that, due to the syntactic restrictions on P_0, the number n of atoms occurring in P_0 are exactly

$$n = 4^k + 4^{k-1} + \ldots + 4 + 1 = \frac{4^{k+1} - 1}{3}$$

from which we can easily calculate

$3 \cdot n + 1 = 4^{k+1}$

i.e.

$\sqrt{3n+1} = 2^{k+1}$

i.e.

$\frac{\sqrt{3n+1}}{2} = 2^k$

Hence we can define the cardinality of $\mathcal{R}_l$ as a function $g(n)$ as follows

$g(n) = f(k) = 4^{\frac{\sqrt{3n+1}}{2} - 1}$.

Let now $x = \frac{\sqrt{3n+1}}{2}$. Then $g(n) = \frac{4^x}{4}$. It is easy to see that, for any constants c, d and sufficiently big x, $4^x > d \cdot x^{2c}$. Hence, for any constant c and sufficiently big n, $g(n) > (\frac{\sqrt{3n+1}}{2})^{2c} = \frac{(3n+1)^c}{4}$.

We can conclude that $g(n)$ is not polynomial in n, the number of atoms of P_0. Hence, the worst case complexity of translating away naf is not polynomial in the number of atoms.

6 Discussion and Conclusions

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.

Negation as failure to find a course of actions is tightly related to negation as failure to prove in logic programming. Indeed, any normal logic program P can be viewed as a GPS $\mathcal{G}_P$ by transforming each rule

$a \leftarrow a_1, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_k$

into a production rule

if $\neg a, a_1, \ldots, a_n,$ *not* $a_{n+1}, \ldots,$ *not* a_k **then assert**(a).

It is not difficult to show that the semantics of P and $\mathcal{G}_P$ coincide, in the sense that each (grounded, preferred, stable) set of computations of $\mathcal{G}_P$ corresponds to a (grounded, preferred, stable) model of P. On the other side, in [8] it has been shown that argumentation can be represented in logic programming with negation as failure. Therefore we can say that the mechanisms of negation as failure to find a course of actions in production systems and negation as failure to prove in logic programming are different sides of the same coin.

There are still several issues which deserve a deeper study and understanding. First, 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 [13, 24]. 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.

Secondly, we intend to study computational mechanisms and proof procedures for general production systems with negation as failure. We have seen that for stratified production systems there is a way to compute negation as failure in a bottom-up fashion. This can be seen as a dynamic or run-time method to compute negation as failure. It is worth addressing the point of whether the class of stratified PS can be extended to more general classes, for which similar bottom-up methods exist. On the other hand, in some cases negation as failure can be compiled down to classical negation, as we have seen in Section 1.2. This can be seen as a compile-time or static method to compute negation as failure. Finding general techniques to achieve this is another interesting issue. As we have already mentioned, in this case negation as failure still serves as a specification for classical (i.e. without naf) production systems.

Finally, we are investigating the application of our approach in the active databases area. Active databases [6] are an important research topic in the database community, due to the fact that they find many applications in real world problems. Many commercial databases systems have been extended to allow the user to express active rules. Still, active databases are faced with many open issues, such as the lack of a well understood declarative semantics (see, e.g. [11, 29]). Such a semantics would provide a common basis for understanding the operational semantics defined by the implementation of active rules in database systems, as well as for comparing different implementations. The typical active rule in active databases is an event-condition-action rule of the form **on** *Event* : **if** *Condition* **then** *Action*. We are currently extending our argumentation based approach to these active rules, and we are addressing also in this case the use of negation as failure in the Condition part of the rules.

References

[1] S. Abiteboul and V. Vianu. Non-determinism in logic-based languages. *Annals of Mathematics and Artificial Intelligence*, 3:151–186, 1991.

[2] A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, 1995.

[3] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[4] A. Bondarenko, P.M. Dũng, R.A. Kowalski, and F. Toni. An abstract, argumentation-based theoretic approach to default reasoning. *Artificial Intelligence*, 93(1-2):63–101, 1997.

[5] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 1-2(109):297–356, 1999.

[6] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan-Kauffman, 1996.

[7] P.M. Dũng. Negation as hypotheses: An abductive foundation for logic programming. In Koichi Furukawa, editor, *Proc. of the Eigth International Conference on Logic Programming*, pages 3–17. The MIT Press, 1991.

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

[9] P.M. Dũng and P. Mancarella. Production systems need negation as failure. In *Proc. AAAI 96*, pages 1242–1247. AAAI Press / The MIT Press, 1996.

[10] P.M. Dũng and T.C. Son. Default reasoning with specificity. In *Proc. First Int. Conf. on Computational Logic*. Imperial College, London UK, 2000.

[11] U. Dayal, E.N. Hanson, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 434–456. Addison-Wesley, 1995.

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

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

[14] H. Geffner and J. Pearl. Conditional entailment: Bridging two approaches to default reasoning. *Artificial Intelligence*, 2-3(53):109–244, 1992.

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

[16] E.N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, 1993.

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

[18] A.C. Kakas and F. Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9(4):515–562, 1999.

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

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

[21] J. L. Pollock. Defeasible reasoning. *Cognitive Science*, 11(4):481–518, 1987.

[22] H. Prakken and G. Sartor. A system for defeasible argumentation, with defeasible priorities. In Michael Wooldridge and Manuela M. Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 1999.

[23] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation. In D. Gabbay, editor, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2000. to appear.

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

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

[26] G. R. Simari and R. P. Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial Intelligence*, 2-3(53):125–157, 1992.

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

[28] G. Vreeswijk. The feasibility of defeat in defeasible reasoning. In J.F. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 526–534, 1991.

[29] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In T. W. Ling, A. O. Mendelzon, and L. Vieille, editors, *Proc. Fourth Int. Conf. on Deductive and Object-Oriented Databases (DOOD 95)*, number 1013 in LNCS, pages 55–72. Springer-Verlag, 1995.

A Appendix

Proof of theorem 4.3

Given a stratified GPS $P = P_0 \cup \ldots \cup P_n$, let L_i be the set of literals l such that $atom(l)$ occurs in the head of some rule in P_i and not in the head of any rule in $P_{i+1} \cup \ldots \cup P_n$.

In the following, if v is a sequence of rules from a GPS P and Q is a subset of P, we denote by $v_{|_Q}$ the sequence obtained from v by dropping all the rules not in Q.

Lemma A.1
Let P be a stratified GPS and let $C \in \mathcal{C}(P)$ Then there exists $C' \in \mathcal{C}(P_0 \cup \ldots \cup P_i)$ such that following properties are satisfied:

(i) $rules(C') = rules(C)_{|_{\bigcup_{k \leq i} P_k}}$ and $initial(C') = initial(C)$

(ii) For each $l \in L_0 \cup \ldots \cup L_i$, $final(C') \models l$ iff $final(C) \models l$

(iii) for each attack $A' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C' there exists also an attack $A \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C such that $rules(A') = rules(A)$.

□

Proof. By induction on the length of C.
The base case, where C is an empty computation, is clear. Let then C be a computation of the form

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

with $n \geq 1$. If $C \in \mathcal{C}(P_0 \cup \ldots \cup P_i)$, there is nothing to prove. Hence, we can assume that there exists a rule $r \in P_{i+1} \cup \ldots \cup P_n$ in $rules(C)$. Let r_j be the rightmost rule in $rules(C)$ such that $r_j \in P_{i+1} \cup \ldots \cup P_n$. Then C has the following form:

$$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_j} S_j \xrightarrow{r_{j+1}} S_{j+1} \ldots \xrightarrow{r_m} S_m$$

where $r_{j+1}, \ldots, r_n \in P_0 \cup \ldots \cup P_i$. Let C_1 be the following computation of length $n-1$:

$$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_{j-1}} S_{j-1} \xrightarrow{r_{j+1}} S'_{j+1} \ldots \xrightarrow{r_m} S'_m$$

where, for each $h \geq j+1$,

$$S'_h = \begin{cases} S_h \setminus \{p\} & \text{if } action(r_j) = \mathbf{assert}(p) \\ S_h \cup \{p\} & \text{if } action(r_j) = \mathbf{retract}(p). \end{cases}$$

The justification for C_1 being a computation is due to the stratification of P. In fact, for each rule r_h with $h \geq j+1$ we have that $r_h \in P_0 \cup \ldots \cup P_{i-1}$ and hence $S'_{h-1} \models cl\text{-}cond(r_h)$ iff $S_{h-1} \models cl\text{-}cond(r_h)$, since neither $\neg p$ nor p can occur in $cl\text{-}cond(r_h)$ due to the stratification of P.

Clearly the following facts hold:

(a) $initial(C_1) = initial(C)$, and

(b) for each $l \in L_0 \cup \ldots \cup L_i$, $final(C_1) \models l$ iff $final(C) \models l$.

Let now $A_1 \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ be an attack against C_1. Let *not* l be the hypothesis occurring in $hyp(r)$, for some r in $rules(C_1)$, such that $final(A_1) \models l$. If r belongs to the sequence $\langle r_1, \ldots r_{j-1}, r_{j+1} \rangle$, clearly A_1 is also an attack against C. Assume now r belongs to the sequence $\langle r_{j+2}, \ldots, r_m \rangle$. Hence, $r = r_i$ for some i such that $j+2 \leq i \leq m$. Let A_1 be of the form

$$S''_0 \xrightarrow{r'_1} S''_1 \ldots \xrightarrow{r'_k} S''_k$$

where $S''_0 = S'_{i-1}$. Since each rule r' in $rules(A_1)$ belongs to $P_0 \cup \ldots \cup P_i$ neither $\neg p$ nor p can occur in $cl\text{-}cond(r')$ due to the stratification of P. Let now, for each h such that $0 \leq h \leq k$

$$S'''_h = \begin{cases} S''_h \cup \{p\} & \text{if } action(r_j) = \mathbf{assert}(p) \\ S''_h \setminus \{p\} & \text{if } action(r_j) = \mathbf{retract}(p). \end{cases}$$

By the previous observation, for each such h, $S'''_h \models cl\text{-}cond(r'_{h+1})$ iff $S''_h \models cl\text{-}cond(r'_{h+1})$. Hence we can construct the following computation $A \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$.

$$S'''_0 \xrightarrow{r'_1} S'''_1 \ldots \xrightarrow{r'_k} S'''_k.$$

Clearly $initial(A) = S_{i-1}$, for some i such that $j+2 \leq i \leq m$ and $final(A) \models l$ such that *not* $l \in hyp(r_i)$. Thus, A attacks C and, by construction, $rules(A) = rules(A_1)$ and so $A \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$. So we can conclude that the following fact also holds:

(c) for each attack $A_1 \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C_1, there exists an attack $A \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C such that $rules(A) = rules(A_1)$.

By applying now the inductive hypothesis to C_1 gives C' satisfying the properties:

(d) $rules(C') = rules(C_1)_{|\bigcup_{k \leq i} P_k}$ and $initial(C_1) = initial(C')$

(e) For each $l \in L_0 \cup \ldots \cup L_i$, $final(C') \models l$ iff $final(C_1) \models l$

(f) For each attack $A' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C' there exists also an attack $A_1 \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C_1 such that $rules(A') = rules(A_1)$.

It is easy to see that the following properties also hold.

- $rules(C') = rules(C)_{|\bigcup_{k \leq i} P_k}$ and $initial(C') = initial(C)$, by (a) and (d)
- for each $l \in L_0 \cup \ldots \cup L_i$, $final(C') \models l$ iff $final(C) \models l$, by (b) and (e)

Let now $A' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ be an attack against C'. Hence, by (f), there exists an attack $A_1 \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C_1 such that $rules(A') = rules(A_1)$. Finally, by (c), there exists an attack $A \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ against C such that $rules(A_1) = rules(A) = rules(A')$, which concludes the proof. $\square$

Lemma A.2
Let P be a stratified GPS. Then $\mathcal{S}(P)$ is a stable set of computations. $\square$

Proof. . By induction on the stratification of P. Let $P = P_0 \cup \ldots \cup P_n$. The base case, $n = 0$, is clear. Let $n \geq 1$. By the inductive hypothesis, it is clear that $\mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$ is conflict-free and attacks each computation in $\mathcal{C}(P_0 \cup \ldots \cup P_{n-1}) \setminus \mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. First we need to prove that $\mathcal{S}(P_0 \cup \ldots \cup P_n)$ is conflict free. Assume the contrary. Let $C, C' \in \mathcal{S}(P_0 \cup \ldots \cup P_n)$ such that C attacks C'. There are two cases:

- $C' \in \mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. Then $C \in \mathcal{S}(P_0 \cup \ldots \cup P_n) \setminus \mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$ since $\mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$ is conflict-free.

  Assume C' has the following form

  $$S_0 \xrightarrow{r_1} S_1 \ldots \xrightarrow{r_m} S_m.$$

  Then there exists h, with $1 \leq h \leq m$ and an assumption *not* $l \in hyp(r_h)$ such that $initial(C) = S_{h-1}$ and $final(C) \models l$. Since $r_h \in P_0 \cup \ldots \cup P_{n-1}$, by the stratification of P, $l \in L_k$ for some $k < n - 1$. By Lemma A.1(i), there exists $C'' \in \mathcal{C}(P_0 \cup \ldots \cup P_k)$ such that:

  - $rules(C'') = rules(C)_{|\bigcup_{j \leq k} P_j}$ and $initial(C'') = initial(C)$.

  Also, by Lemma A.1(ii) and the fact that $l \in L_k$, we have that $final(C'') \models l$ iff $final(C) \models l$, hence $final(C'') \models l$. So, C'' also attacks C' and clearly $C'' \in \mathcal{C}(P_0 \cup \ldots \cup P_{n-1})$.

  Therefore, C'' is attacked by one computation in $\mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. From lemma A.1(iii), C is attacked by $\mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. This contradicts the fact that $C \in \mathcal{S}(P_0 \cup \ldots \cup P_n)$.

- $C' \in \mathcal{S}(P_0 \cup \ldots \cup P_n) \setminus \mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. By definition of $\mathcal{S}$, C must also belong to $\mathcal{S}(P_0 \cup \ldots \cup P_n) \setminus \mathcal{S}(P_0 \cup \ldots \cup P_{n-1})$. Let *not* l be a hypothesis underlying C' such that $final(C) \models l$. Therefore, due to the stratification of P, $l \in L_k$ with $k < n$.

  As in the previous case, by exploiting Lemma A.1, we can find $C'' \in \mathcal{C}(P_0 \cup \ldots \cup P_k)$ such that $initial(C) = initial(C'')$ such that C'' attacks C'.

  We claim that $C'' \in \mathcal{S}(P_0 \cup \ldots \cup P_k)$. Assume the contrary. Since, by the inductive hypothesis, $\mathcal{S}(P_0 \cup \ldots \cup P_k)$ is stable, C'' is attacked by $\mathcal{S}(P_0 \cup \ldots \cup P_k)$. By Lemma A.1(iii) C is also attacked by $\mathcal{S}(P_0 \cup \ldots \cup P_k)$, and this contradicts the fact that $C \in \mathcal{S}(P_0 \cup \ldots \cup P_n)$. So, $C'' \in \mathcal{S}(P_0 \cup \ldots \cup P_k)$, and C'' attacks C', which is again a contradiction to the fact that $C' \in \mathcal{S}(P_0 \cup \ldots \cup P_n)$.

Now we want to show that $\mathcal{S}(P_0 \cup \ldots \cup P_n)$ attacks every computation in $\mathcal{C}(P_0 \cup \ldots \cup P_n) \setminus \mathcal{S}(P_0 \cup \ldots \cup P_n)$. This is clear from the definition of $\mathcal{S}$. □

Lemma A.3
Let P be a stratified GPS. Then $\mathcal{S}(P)$ is grounded. □

Proof. First, we want to show that for each $i \geq 0$, $\mathcal{S}(P_0 \cup \ldots \cup P_{i+1}) \subseteq \mathit{Def}(\mathcal{S}(P_0 \cup \ldots \cup P_i)$.

Let $C \in \mathcal{S}(P_0 \cup \ldots \cup P_{i+1})$ and let C' be an attack against C. Let $C'' \in \mathcal{S}(P_0 \cup \ldots \cup P_i)$ defined following lemma A.1 with respect to C'. Hence from lemma A.1(i,ii), C'' attacks C. Therefore from the definition of $\mathcal{S}$, $C'' \notin \mathcal{S}(P_0 \cup \ldots \cup P_i)$. Hence C'' is attacked by $\mathcal{S}(P_0 \cup \ldots \cup P_i)$. From lemma A.1(iii), $\mathcal{S}(P_0 \cup \ldots \cup P_i)$ attacks C'. That means that C is defended by $\mathcal{S}(P_0 \cup \ldots \cup P_i)$.

It is easy to see that $\mathcal{S}(P_0) \subseteq \mathit{Def}(\emptyset)$, it follows that $\mathcal{S}(P) \subseteq \bigcup_{i \geq 0} \mathit{Def}^i(\emptyset)$. From the fact that $\mathcal{S}(P)$ is stable, it follows immediately that $\mathcal{S}(P)$ is also grounded. □

The main theorem 4.3 is now a simple corollary of the previous lemmas.

Proof of theorem 5.5
(i) Assume that C has the following form

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

Let us construct C' as follows

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

Clearly C' is a possible computation of P, since for $0 \leq i < n$, $S \models \mathit{cl\text{-}cond}(r_i)$, due to the fact that $S \models \mathit{cl\text{-}cond}(r_i^*)$ and $\mathit{cl\text{-}cond}(r_i) \subseteq \mathit{cl\text{-}cond}(r_i^*)$. We need to show that there is no attack against C'. Assume the contrary. Hence, for some $i > 0$, there is *not* $l \in \mathit{hyp}(r_i)$ and a possible computation C'' such that $\mathit{initial}(C'') = S_{i-1}$, and l holds in $\mathit{final}(C'')$. Assume $l \in L_k$: by the fact the P is stratified, it is clear that r_i belongs to a stratum P_h such that $h > k$. By Lemma A.1 there exists a computation $C''' \in \mathcal{C}(P_0 \cup \ldots \cup P_k)$ such that

- $\mathit{rules}(C''') = \mathit{rules}(C'')_{| \bigcup\limits_{m \leq k} P_m}$ and $S_{i-1} = \mathit{initial}(C'') = \mathit{initial}(C''')$
- For each $l' \in L_0 \cup \ldots \cup L_k$, $\mathit{final}(C''') \models l'$ iff $\mathit{final}(C'') \models l'$

Assume C''' is the empty computation. Hence, $\mathit{initial}(C''') = \mathit{final}(C''') = S_{i-1}$ and $S_{i-1} \models l$. This contradicts the fact that $S_{i-1} \models l^*$.

Hence C''' is not empty. Let C''' have the following form:

$$S_{i-1} \xrightarrow{r} S' \ldots \longrightarrow.$$

Since r belongs to a lower stratum than r_i, r^* has a higher priority than r_i^*. Therefore r_i^* is not applicable in S_{i-1}. This is a contradiction to the fact that C is a computation in $\langle P^*, \prec \rangle$.

(ii) Let C be a complete computation of $\langle P^*, \prec \rangle$. Hence there is no rule r^* which is applicable in *final*(C). Let C' be the computation of P constructed as in (i). For each rule r we have two cases:

- *final*$(C') \not\models cl_c ond(r)$, hence r is not applicable in *final*(C').
- *final*$(C') \models cl_c ond(r)$, hence r is possibly applicable in *final*(C'). Since r^* is not applicable in *final*(C), there exists a naf literal *not* l in $hyp(r)$ such that *final*$(C) \models \neg l^*$. Hence, the possible computation *final*$(C) \xrightarrow{r} S'$ is attacked by the empty computation.

This means that, if C is a complete computation, C' is also complete.

Proof of theorem 5.16

It is important to notice that, for each partial computation C, if an atom occurs in one (incomplete) state of C[3], then it occurs in all states of C.

We first prove that l is achievable from each $label(N)$ by induction on the length of the path from node N to the root.

Base case: obvious.

Inductive case: let M be the parent of N, r be the rule labeling the link from N to M , and l be achievable from $label(M)$ by inductive hypothesis. Hence, there exists a partial computation C_M such that $initial(C_M) = label(M)$ and $final(C_M) \models l$. Let J be the set consisting of those literals in $label(N)$ whose atom do not occur in the C_M. Let C be the following partial computation

$$label(N) \xrightarrow{r} label(M) \cup J.$$

Let C'_M be the partial computation obtained from C_M by replacing every incomplete state I in C_M by $I \cup J$. From the definition of J, no atom occurring in J occurs in I, hence C'_M is obviously a partial computation. It is now easy that the concatenation of C and C'_M is a partial computation in P_0.

Now we have to prove that, for each incomplete state I_0 such that l is achievable from I_0 in P_0, $I_0 \supseteq label(N)$, for some node N in $T_{P_0,l}$. Let

$$I_0 \xrightarrow{r_1} I_1 \xrightarrow{r_2} \dots I_{n-1} \xrightarrow{r_n} I_n$$

We prove the result by induction on n

Base case: $n = 0$ obvious.

Inductive case: since l is achievable from I_1 by a partial computation of length $n-1$, by inductive hypothesis there exists a node M such that $label(M) \subseteq I_1$. There are two cases:

[3]An atom p occurs in an incomplete state I if either $p \in I$ or $\neg p \in I$.

- M has a child N linked to M by r_1. By definition of $T_{P_0,l}$ and of partial computation we have:
  - if $action(r_1) = \textbf{assert}(p)$ then $p \in label(M)$ and $label(N) = cond(r_1) \cup label(M) \setminus \{p\}$ and $I_1 = I_0 \setminus \{\neg p\} \cup \{p\}$. Obviously, $cond(r_1) \subseteq I_0$, and $label(M) \setminus \{p\} \subseteq I_1 \setminus \{p\} \subseteq I_0$. Hence $label(N) \subseteq I_0$.
  - if $action(r_1) = \textbf{retract}(p)$ then $\neg p \in label(M)$ and $label(N) = cond(r_1) \cup label(M) \setminus \{\neg p\}$, and $I_1 = I_0 \setminus \{p\} \cup \{\neg p\}$. Obviously, $cond(r_1) \subseteq I_0$, and $label(M) \setminus \{\neg p\} \subseteq I_1 \setminus \{\neg p\} \subseteq I_0$. Hence $label(N) \subseteq I_0$.
- M has no child N linked to M by r_1. By definition of $T_{P_0,l}$ and of partial computation we have:
  - if $action(r_1) = \textbf{assert}(p)$ then neither p nor $\neg p$ occur in M, $I_1 = I_0 \setminus \{\neg p\} \cup \{p\}$. Hence $label(M) \subseteq I_0$.
  - if $action(r_1) = \textbf{retract}(p)$ then neither p nor $\neg p$ occur in M, $I_1 = I_0 \setminus \{p\} \cup \{\neg p\}$. Hence $label(M) \subseteq I_0$.

$\square$

Biography of the authors

Phan Minh Dũng received both his Master and PhD degrees in Computer Science from the Dresden University, Germany in 1981 and 1984, respectively. He is currently full professor and coordinator of the Department of Computer Science and Information Management, School of Advanced Technology, Asian Institute of Technology (AIT), Bangkok, Thailand. His research interests include computational logics, logic programming, knowledge representation, multiagent programming and computer security.

Paolo Mancarella received the Laurea Degree in Computer Science from University of Pisa, Italy, in 1982, and the PhD degree in Computer Science from University of Pisa, Italy, in 1988. He is currently Associate Professor in Computer Science at the Department of Computer Science, University of Pisa, Italy. His research interests include semantics of programming languages, logic programming, knowledge representation, computational logic, deductive and spatio–temporal databases.

Contact Information

Phan Minh Dũng
Department of Computer Science and Information Management
Asian Institute of Technology
GPO Box 2754
Bangkok 10501
Thailand
e-mail: dung@cs.ait.ac.th
Ph. +66 2 524 5709
Fax. +66 2 524 5721

Paolo Mancarella
Dipartimento di Informatica
Universitá di Pisa
Corso Italia, 40
56125 Pisa
Italy
e-mail: p.mancarella@di.unipi.it
Ph. +39 050 887210
Fax. +39 050 887226

Corresponding author

Paolo Mancarella