Research Report

Determinism in Partially Ordered Production Systems

Joseph M. Hellerstein
IBM Almaden Research Center
San Jose, CA 95120-6099
hellerst@ibm.com

Meichun Hsu
Aiken Computation Lab
Harvard University, Cambridge, MA 02138
hsu@harvard.harvard.edu

Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

Determinism in Partially Ordered Production Systems

Joseph M. Hellerstein
IBM Almaden Research Center
San Jose, CA 95120-6099
hellerst@ibm.com

Meichun Hsu
Aiken Computation Lab
Harvard University, Cambridge, MA 02138
hsu@harvard.harvard.edu[1]

March, 1991

[1]Current address: Digital Equipment Corp., 800 West El Camino Real, Mountain View, CA 94040

Abstract

The subtlety of interactions between rules in a production system has motivated research into ways of making production systems behave more predictably. In this paper we generalize two important inter-rule relations and use them to develop a new class of production systems that are deterministic, in the sense that they have the *unique fixed point property* (**ufp**). This class, the *implied interfence ordered* (**iio**) systems, is shown to subsume previously identified systems with **ufp**, including stratified systems [2, 7] as well as accumulative interference-free systems [13]. Although the **iio** systems depend on a production ordering conflict resolution scheme, we present an algorithm to convert any production system with production ordering into a computationally equivalent system without ordering, which makes our result more universally applicable, and points to the possibility of developing a model theory for the **iio** systems.

1. Introduction

A variety of branches of computer science have become interested in the concept of a rule base system as a method of computation. Much heuristic research has been devoted to developing rule base systems particularly suited to specific tasks. One result of this research is that rule bases have proven to be much more complex and unwieldy than had been expected [4]. Even small rule bases may have extremely subtle interactions between rules, making it difficult to maintain and extend such systems.

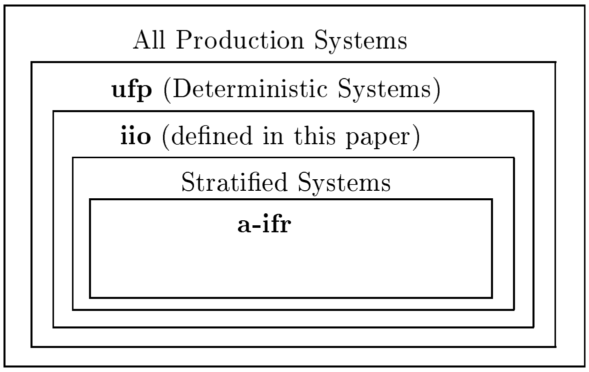
With this in mind, it becomes interesting to narrow the focus and consider only those rule base systems that are predictably deterministic in their action. A successful model for rule base implementation is that of *production systems*, which include the workings of an interpreter as an explicit part of the model. We thus examine such systems, and ways in which they can be rendered deterministic. Although not necessarily appropriate for all applications of production systems, deterministic production systems free their creators from the many complications which arise in production systems that are not deterministic, and thus such systems are more easily understood, maintained and extended.

Deterministic production systems may prove particularly useful for environments that couple rule bases and an interpreter to traditional databases. Typically, such systems are developed to encourage greater automation by having the rule base carry out obvious consequences as part of each transaction. In these environments deterministic action is extremely useful, since the purpose of a rule base in such a situation is to automate deterministic activity which would otherwise have to be recognized as necessary by the user, and then executed by hand.

1.1. Overview

This paper develops a new class of production systems, *implied-interference-ordered* (**iio**) systems, and demonstrates that they are deterministic, in the sense that they have the *unique fixed point property* (**ufp**). The set of **iio** systems is shown to subsume previously identified classes of deterministic rule systems, including *accumulative interference-free* (**a-ifr**) [13] systems, as well as *stratified production systems* [2, 7]. Finally, although the set of **iio** systems is defined by an ordering on the productions in the system, it is shown that systems with orderings can be converted to computationally equivalent systems without orderings, thus maintaining the utility of **iio** systems for environments that do not allow production ordering. This paper does not consider issues of production system termination.

A hierarchy of known classes of production systems is shown in the following diagram:



1.2. Structure of the Paper

Section 2 provides a background on production systems in general, and on the systems we will consider here. Section 3 defines the basic inter-production relationships which are the key to the later results, and clarifies what we mean by a deterministic production system. Section 4 defines the class of **iio** systems, and proves them to be deterministic, while Section 5 presents the class of production memories that are subject to **iio**. Section 6 shows that stratified production systems form a subset of **iio** systems. Section 7 demonstrates the applicability of our results to systems without production ordering, by providing a construction for taking an ordered production system and building a computationally equivalent unordered system. Section 8 discusses the significance of our findings and the implications for future research.

2. Background

A *production system* consists of three parts: a database, a set of rules or *productions*, and an interpreter. The database is sometimes referred to as *working memory*, and the set of productions as *production memory* [5].

2.1. Working Memory: The Database

The database may simply be viewed as a finite set of units of information, not necessarily of similar forms. We call the universe of the database ω, and define S_ω to be the set of finite subsets of ω:

$$S_\omega = \{\, x \mid (x \subseteq \omega) \wedge (|x| < \infty) \,\}.$$

Elements of S_ω are called *states* of the database. Inifinite states of the database are not considered due to their lack of practical applicability. Unless otherwise stated, we will assume that a given production system has a database over some unspecified universe ω.

2.2. Production Memory

The definition of a production is derived from the definition in [13] of a rule in a *rule triggering system*, although the concept of instantiations described below is due to [5].

Definition 1. *A* production P *is a* condition-action *rule of the form*

$$P : \delta v_1, v_2, \ldots, v_m.F_P \rightarrow action(A_1, A_2, \ldots, A_n)$$

where F_P is a first order formula which is safe *(i.e., not true on infinitely many tuples in a finite Herbrand interpretation, cf. [8]); action is either* **insert** *or* **delete**[1]*; $\delta v_1, v_2, \ldots, v_m$ is the* δ-prefix*, a list of the free variables in F_P and in $A_1, A_2, \ldots, A_n$; and $A_1, A_2, \ldots, A_n$ is a set of elements in ω, including free variables that form a subset of the δ-prefix.*

We also impose the restriction that for a production $P : \delta v_1, v_2, \ldots, v_m.F_P \rightarrow$ **delete**$(A_1, A_2, \ldots, A_n)$*, F_P must have as one of its outer conjuncts the disjunction $(A_1 \vee \ldots \vee A_n)$. This does not restrict the power of the language, as we shall see below, but facilitates the proof of our major result. This restriction also exists in the OPS5 language [7].*

The δ-prefix serves to link the free variables in F_P with those in the elements of the right-hand-side of P. The action of a production specifies whether these elements are to be inserted into or deleted from the database.

2.3. The Interpreter

The interpreter of a production system works in a *forward-chaining, recognize-act* cycle, examining the database to find elements that satisfy the productions in the system, and then carrying out the associated actions. Different ways of satisfying the condition of a production may result in the production having different actions. The following definitions are useful at this point:

[1]This is a typical scheme, and is used in [7]. Our results in this paper should also be applicable to systems that allow combinations of insertion, deletion and/or update actions in a single rule.

Definition 2. [2] *A substitution* $\theta \equiv (t_1/v_1)\dots(t_m/v_m)$ *replaces all free ocurrences of the variables* $v_1, \dots, v_m$ *in a production with terms* $t_1, \dots, t_m$ *respectively.* $F\theta$ *is the result of applying the substitution* θ *to the formula* F*, and is called an* instance *of* F*. The null substitution* $\theta_\emptyset$ *can be applied to formulas with no free variables, and for any such formula* F, $F\theta_\emptyset \equiv F$.

Definition 3. *An* instantiation *of a production* $P : \delta v_1, v_2, \dots, v_m.F_P \rightarrow \mathit{action}(A_1, A_2, \dots, A_n)$ *is an ordered pair* $I =< P, \theta >$ *such that instance* $F_P\theta$ *is satisfied over the current state of the database. The actions of* I *are* $\mathit{action}(A_1\theta, A_2\theta, \dots, A_n\theta)$*, i.e. the result of applying* θ *to the action elements of* P.

An instantiation $< P, \theta >$ is said to be *non-trivial* if $\mathit{action}(A_i\theta)$ makes a change in the current state of the database for some $A_i\theta$; for an **insert**, this means that at least one $A_i\theta$ does *not* exist in the database. The restriction on our syntax removes the possibility of trivial instantiations of productions with **delete** actions. Note that the restriction has no other affect on the language than to eliminate trivial instantiations (which take no action), and thus it does not restrict the power of the language in any way.[2]. We denote the set of non-trivial instantiations of a production P at state x as $\text{INS}_P(x)$. If $\text{INS}_P(x) \neq \emptyset$, then we say that P is *instantiated* or *triggered* at x. (From this point on, we will use the term *instantiation* as a shorthand for *non-trivial instantiation.*)

The 'recognize' phase of an interpreter cycle involves finding all the (non-trivial) instantiations of the productions in the system. Note that multiple instantiations may exist for a given production in any interpreter cycle.

Example 1. Throughout this paper, we shall use the same production system for our examples: $\omega = \{$(Tom, CAT), (Jerry, MOUSE), (Mickey, MOUSE), (Fido, DOG), (CHEESE)$\}$, the working memory is $\{$(Jerry, MOUSE), (Mickey, MOUSE), (Tom, CAT)$\}$, unless otherwise specified, and the production memory consists of the following three productions

$$P_1 : \delta.\neg(\text{Jerry}, \text{MOUSE}) \wedge (\text{CHEESE}) \rightarrow \textbf{insert}((\text{Mickey}, \text{MOUSE}))$$
$$P_2 : \delta v_1.(v_1, \text{MOUSE}) \rightarrow \textbf{insert}((\text{Fido}, \text{DOG}))$$
$$P_3 : \delta v_1, v_2.\neg(\text{Fido, DOG}) \wedge (v_1, \text{MOUSE}) \wedge (v_2, \text{CAT}) \rightarrow \textbf{delete}((v_1, \text{MOUSE}))$$

The instantiations for this system are $< P_2$, (Mickey/v_1) $>$, $< P_2$, (Jerry/v_1) $>$, $< P_3$, (Mickey/v_1)(Tom/v_2) $>$, and $< P_3$, (Jerry/v_1)(Tom/v_2) $>$. P_1 has no instantiations in this state of the working memory.

The *conflict set* of a production system Γ with production memory F is defined at state $x \in S_\omega$ to be $\bigcup_{P \in F} \text{INS}_P(x)$, i.e. the set of instantiations found in the 'recognize' phase of the interpreter cycle at x. This set is unique for every state $x \in S_\omega$ of the database. The interpreter uses a *conflict resolution* scheme to decide which instantiations in the conflict set should be *fired*, that is, have their actions carried out during the 'act' part of the interpreter cycle. In the systems we will consider, all instantiations of exactly one production are fired on a given cycle, and the default conflict resolution scheme is to fire the instantiations of an arbitrary production that has been triggered[3]. We call such a conflict resolution scheme an *Arbitrary Decision* scheme[4] [5]. Since Arbitrary Decision schemes choose productions entirely nondeterministically, and since they are nearly always incorporated only as the last step in any conflict resolution scheme (c.f. [5]) a system which makes exclusive use of an Arbitrary Decision scheme is considered to have no conflict resolution at all, and is termed a *conflict-resolution-free* (*CR-free*) system.

The class of conflict resolution schemes that will interest us in this paper are *production ordering* schemes [5], which place a partial order $\prec$ on the productions of the system. When the conflict set contains instantiations $I_1 \dots I_m$ resulting from production P, and instantiations $I_{m+1} \dots I_n$ resulting from production Q, and $P \prec Q$, then the interpreter may choose to perform the actions of $I_1 \dots I_m$ on this cycle, but will not perform the actions of $I_{m+1} \dots I_n$. The effect of a production ordering scheme, then, is to filter

[2]It should be noted at this point that working memory duplicates are not considered here. If a production specifies the insertion of an element that already exists in the database, that part of the production's action is not carried out. Similarly, if a production specifies the deletion of an element that does not exist in the database, that part of the production's action is ignored.

[3]In these systems we speak of *productions* being fired, rather than instantiations. Firing production P at state x means firing all instantiations in $\text{INS}_P(x)$.

[4]Note that McDermott and Forgy's 'AD1' differs slightly but significantly from the Arbitrary Decision scheme presented here — it chooses a single arbitrary *instantiation*, rather than a single arbitrary production.

the conflict set by excluding instantiations of those productions in the conflict set that are 'dominated' by other productions in the conflict set. After this is done, the remainder of the conflict set is passed on to an Arbitrary Decision scheme.

2.4. Database States and Transition Relations

It will prove useful to describe the computations of a production system using the traditional model of states and transitions. Consider a production system Γ with production memory F. We may regard a production as a function which maps $S_\omega \to S_\omega$ such that for $P \in F$, $P(x) = x$ if P is not triggered in x or if the conflict resolution of Γ prevents P from firing in x. Otherwise, $P(x)$ is considered to be the state reached after carrying out the actions of all instantiations in $\text{INS}_P(x)$.

A state x is called a *fixed point* of a production system if for every production P in the system $P(x) = x$. We denote the set of fixed points of a production system Γ as $\mathbf{fp}(\Gamma)$. Note that our definition of the interpreter cycle requires *exactly* one production to be chosen from the conflict set, and therefore a fixed point is reached only when no productions are triggered.

We call a computation $x \xrightarrow{P} y$ *effective* if $y \neq x$. The effective computation steps of a production system Γ with production memory F define a transition relation $\to_\Gamma : S_\omega \times S_\omega$ called the *transition relation associated with* Γ: $x \to_\Gamma y \iff \exists P \in F \;\; y = P(x) \neq x$. Let $\xrightarrow{*}_\Gamma$ be the *reflexive* and *transitive* closure of $\to_\Gamma$. A transition sequence starting at $x_0 \in S_\omega$ is said to *terminate* in $x_n \in S_\omega$ iff $x_n \in \mathbf{fp}(\Gamma)$. If it terminates in x_n we also say that x_n is a fixed point *reachable* from x_0. We call a sequence of transitions from a given state x to another state y a *computation path* from x to y.

2.5. Related Work

As discussed below, Zhou and Hsu define the *unique fixed point* property of rule bases, and prove that an *accumulative interference-free* rule base has this property [13]. Apt, Blair and Walker [2] identify *stratified* logic programs, and Raschid [7] extends these results to develop a definition of stratified production systems. Stratified systems also have the unique fixed point property, and are subsumed by the **iio** systems developed below.

McDermott and Forgy [5] provide a description of various conflict resolution strategies (including various production ordering schemes) with a qualitative discussion of their utility and appropriateness to different applications. Jackson [4] summarizes and comments on this work, and describes the conflict resolution schemes available for use in the OPS5 production system language.

Some examples of production systems, both elementary and complex, are discussed in [4], [11], [6], [10], etc. A number of papers relevant to production systems are collected in [9].

3. Properties of Production Systems

Some definitions are useful in order to formalize the interactions between productions. Two basic inter-production relationships are defined here, and used to build an intuitive and easily examined dependency graph which comes into play in our proofs below.

3.1. The Gives-To Relation

The first inter-production relationship we need to capture is a causal one: we wish to define when firing one production can cause another to have new instantiations.

Definition 4. *For 2 productions P and Q in a production system, we say that P* gives to *Q ($P \to Q$) if*

$$\exists x \in S_\omega \;\; \text{INS}_Q(P(x)) - \text{INS}_Q(x) \neq \emptyset,$$

that is, if for some state x of the database, firing P produces a previously unsatisfied instantiation of Q.

Example 2. In our example system, we observe that $P_1 \to P_2$, by considering the working memory state $x = \{(\text{CHEESE})\}$: $\text{INS}_{P_2}(x) = \emptyset$, but $\text{INS}_{P_2}(P_1(x)) = \{< P_1, (\text{Mickey}/v_1) >\}$.

Note that our restriction on the syntax of **delete** productions has a side effect in terms of the gives-to relation. Given two productions, P_1 which can insert some element k, and P_2 which can delete element k, $P_1 \to P_2$, since F_{P_2} will include k (or a variable for which k is a substitution) in one of its outer conjuncts. This observation will prove important in a later proof.

Example 3. In our example system, we see that $P_1 \to P_3$, since for state $x = \{(\text{Tom, CAT})\ (\text{CHEESE})\}$, $\text{INS}_{P_3}(x) = \emptyset$, but $\text{INS}_{P_3}(P_1(x)) =< (\text{Mickey}/v_1)(\text{Tom}/v_2) >$. This is a result of the restriction on **delete** rules. If P_3 did not have (v_1, MOUSE) as one of its conjuncts, then it would not have been true that $P_1 \to P_2$.

We will later be interested in $\xrightarrow{*}$, the *reflexive* and *transitive* closure of the gives-to relation, because if in some state x there is no production P triggered such that $P \xrightarrow{*} Q$, then we can be assured that Q will never be triggered. (The gives-to relation is not to be confused with the transition relation associated with production system Γ, which is always denoted in the text as $\to_\Gamma$ in general, or $\xrightarrow{P}$ for a transition on a specific production P.)

3.2. The Interference Relation

The counterpart to the gives-to relation is called *interference*, and can be roughly described as the situation where firing a production P can prevent the firing of some potential instantiations of another production Q. That is, interference captures a lack of monotonicity in the actions of productions. Interference is a subtler relation than gives-to, however, so we must be careful in our definition. Some background is necessary before the discussion can continue.

Definition 5. *The* action set *of a production P at state x is defined as* $\text{AS}_P(x) = [A_P(x), D_P(x)]$ *where*

$$A_P(x) = P(x) - x, \quad D_P(x) = x - P(x)$$

i.e. $A_P(x)$ is the assertion set, elements which the production is capable of inserting into the database at x, and $D_P(x)$ is the deletion set, elements which the production is capable of deleting from the database at x. We define the global action set *of P as* $\text{AS}_P = [A_P, D_P]$ *where*

$$A_P = \bigcup_{x \in S_\omega} A_P(x), \quad D_P = \bigcup_{x \in S_\omega} D_P(x).$$

Note that the global action set contains the actions of *all* possible instantiations of a production. Thus if one of the action elements of a production contains a free variable, the global action set of that production includes all possible substitutions for the variable over ω. Note further that due to the production semantics considered here, one of A_P or D_P is always $\emptyset$ for any global action set (and therefore one of $A_P(x)$ or $D_P(x)$ is $\emptyset$ for any production at any state x). Action sets are a variation of the *affected sets* of [10].

Definition 6. *Given production P with global action set $\text{AS}_P = [A_P, D_P]$ and a production Q with global action set $\text{AS}_Q = [A_Q, D_Q]$, we say that Q is* monotonic in AS_P *(notation: $Q \mid_m \text{AS}_P$) iff*

$$\forall x, y \in S_\omega \ \ ((y - x) \subseteq A_P) \wedge ((x - y) \subseteq D_P) \Longrightarrow A_Q(x) \subseteq Q(y) \wedge D_Q(x) \cap Q(y) = \emptyset$$

that is, any element added by Q at x appears in $Q(y)$, and any element deleted by Q at x does not appear in $Q(y)$, i.e. the set of effects of actions which Q would take grows as actions within the action set of P are carried out.

This definition expands the definition of *local monotonicity* in [13], allowing us to expand the interference defined in [13] to encompass productions with deletion actions:

Definition 7. *A production P* interferes with *another production Q ($P \twoheadrightarrow Q$) if Q is not monotonic in the action set of P ($Q \not|_m \text{AS}(P)$).*

This is the relation that provides the desired contrast to the gives-to relation: if $P \rightarrow Q$, then firing P can cause Q to take more action than it would have otherwise; if $P \twoheadrightarrow Q$, then firing P can cause Q to take less action than it would have otherwise. Discovering interferences may be difficult, however. Intuitively, we see that if an element $k \in \omega$ is inserted in the action of a production P, then P will interfere with any production Q where the condition of Q may be satisfied by the absence of k from working memory. Similarly, if an element $k \in \omega$ is deleted by the action of a production P, then P will interfere with any production Q where the condition of Q may be satisfied by the presence of k in working memory.

Example 4. In our example system, we see that $P_2 \twoheadrightarrow P_3$, because $\textsc{as}(P_2) = [\{(\text{Fido, DOG})\}, \emptyset]$, and it is clear that the action set of P_3 does not grow monotonically as you add "Fido the dog" to the working memory; if P_3 had been triggered when P_2 was fired, none of the instantiations of P_3 would be satisfied after the firing of P_2.

Given the assertion above that interference is a relation contrasting to gives-to, one might well ask why interference was not defined as the exact converse of gives-to, i.e. $P \twoheadrightarrow Q$ if $\exists x \in S_\omega$ such that $\textsc{ins}_Q(x) - \textsc{ins}_Q(P(x)) \neq \emptyset$. The problem with this shorter definition is that it is not sufficiently restrictive; for instance, by this definition, every production interferes with itself, which would preclude the accumulative interference-free systems of [13], which we discuss below. The definition we use is an extension of the definition in [13], and better suited to our purposes.[5]

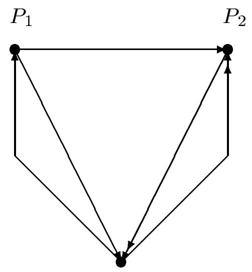
3.3. Production Memory Dependency Graphs

It will prove useful, given the definitions of the gives-to and interference relations, to construct a graph representing the production memory of a system and its interrelationships.

Definition 8. *A* dependency graph *for a production memory F is a directed graph in which the vertices $v_1, \ldots, v_n$ represent the productions $P_1, \ldots, P_n$ in F respectively, and the edges are partitioned into two classes as follows:*

1. *A* gives-to edge (v_i, v_j) *is in the graph if $P_i \rightarrow P_j$ in F. Pictorially this is represented as:* $\overset{P_i}{\bullet} \longrightarrow \overset{P_j}{\bullet}$.
2. *An* interference edge (v_k, v_l) *is in the graph if $P_k \twoheadrightarrow P_l$ in F. Pictorially this is represented as:* $\overset{P_k}{\bullet} \twoheadrightarrow \overset{P_l}{\bullet}$.

Example 5. The dependency graph for our example system appears below.



3.4. Production Systems With the Unique Fixed Point Property

A production system is considered to be deterministic if it has the *unique fixed point property*, which requires that from any state of the database the system will reach a unique fixed point, regardless of the order in which the productions are fired.

[5]It can be argued that the definition of gives-to is equivalent to the converse of the definition of interference, although this argument is rather lengthy. Thus we could have forced the definition of the gives-to relation to mirror the interference relation definition, but it was judged preferable to keep the gives-to definition as concise and intuitive as possible.

Definition 9. [12] *A production system* Γ *has the* unique fixed point property (**ufp**) *if*

$$\forall x, y \in S_\omega\ (x \xrightarrow{*}_\Gamma y) \wedge (x \xrightarrow{*}_\Gamma z \in \mathbf{fp}(\Gamma)) \Longrightarrow y \xrightarrow{*}_\Gamma z.$$

An *accumulative* production system is one in which all actions are **insert** actions. Termination is guaranteed for an accumulative production system, but not for production systems in general. Thus in choosing **ufp** as our measure of determinism, we may be considering some production systems to be "deterministic" in their action without knowing whether they are guaranteed to terminate. Thus when we call a production system deterministic, we mean that it is guaranteed not to have an indeterminate termination state.

Zhou and Hsu prove that an accumulative, interference-free (**a-ifr**) production system has **ufp** [13]. The reason that interference can produce nondeterminism is readily seen. Recall that if $P \twoheadrightarrow Q$, then firing P can prevent some potential instantiations of Q from being satisfied. Now if P and Q are triggered in the same interpreter cycle, and there is no conflict resolution specifying which to fire first, then the outcome of the firing can be non-deterministic: if P is fired before Q, then Q can have fewer instantiations than if Q had been fired before P.

Example 6. Consider our example system, but with working memory {(Mickey, MOUSE), (Tom, CAT)}. Both P_2 and P_3 are triggered in this state, and we know from the previous example that $P_2 \twoheadrightarrow P_3$. If we fire P_3 first, Mickey gets 'eaten' (i.e. deleted) and we reach a fixed point with working memory {(Tom, CAT)}; if we fire P_2 first we reach a fixed point with working memory {(Fido, DOG), (Mickey, MOUSE), (Tom, CAT)}.

4. Production Ordering and ufp: Implied Interference Ordering

If a production system has **ufp**, then no conflict resolution scheme will have an effect on the fixed point reached by the system. Thus an **a-ifr** production system remains unaffected by conflict resolution.

If a production system is not interference-free, however, then conflict resolution such as production ordering may indeed have a bearing on the fixed point that will be reached from any given state of the database. This results from the possibility of a production ordering bringing determinism to the aformentioned non-determinisms arising from interference.

Consider a system in which there is only one interference, $P \twoheadrightarrow Q$. We may think that ordering $P \prec Q$ would make the system deterministic, since it removes the nondeterminism in the situation where both P and Q are triggered. However, consider the fact that Q is now triggered only when P is not triggered. Any other production $R \to P$ now can prevent the firing of Q, since firing R may trigger P, preventing Q from firing. The converse problem arises if we order $Q \prec P$; in this case, any $S \to Q$ can prevent P from firing. The issue in both cases is identical. To examine the case where $P \prec Q$, the interaction between R and Q may be seen as a sort of weak interference, since firing R can postpone the firing of Q, thus affecting the computation of the system. The same can be said of S and P in the case where $Q \prec P$. We call such interactions that arise from the ordering *implied interferences* of the ordering, and later we shall see that they correspond to actual interferences if the system is converted to a computationally equivalent CR-free system.

It appears intuitive, at this point, that if we can order the new implied interferences the same way we ordered the interferences that implied them, and repeat the process until no new implied interferences arise, then we can force **ufp** on the system. This is proven to be true when applicable, but we first define the class of orderings that arise from this argument, and then show which systems can have such orderings.

Definition 10. *A production system with production memory F is called* implied-interference-ordered (**iio**) *if for two productions P, $Q \in F$ such that $P \twoheadrightarrow Q$ either $P \prec Q$ or $Q \prec P$, and for any productions R, S, T if $R \xrightarrow{*} S$ and $S \prec T$, then $R \prec T$.*

The name of the class comes from the above concept of implied interferences, since it ensures that a consistent ordering exists of interferences and their resulting implied interferences. We shall use **iio** to refer to the adjectival form (*implied-interference-ordered*) as well as the noun form (*implied-interference-ordering*). We now proceed to prove our desired result, demonstrating **iio** systems as a new class of systems with **ufp**:

Lemma 1. *Consider a production system Γ with production memory F having productions $P, Q \in F, P \neq Q$ such that $P \prec Q$. If Γ is* **iio** *then if P and Q fire in any computation path from a state x, they will fire in order — that is, P will never fire again once Q fires.*

Proof. Since Γ is **iio**, Q cannot be fired while any production $R \overset{*}{\to} P$ is triggered. Therefore, at the time Q is fired P is not triggered and there is no production $R \overset{*}{\to} P$ triggered, and thus P will not fire again, so P fires only before Q. □

Lemma 2. *Given an* **iio** *production system Γ, a state $x \in S$, and two computation paths out of x, each ending in a fixed point, it is guaranteed that the set of instantiations fired during each path will be identical.*

Proof. We prove that the set of instantiations fired in the second path is a subset of the set of instantiations fired in the first path, by induction on the number of interpreter cycles in the second path. The converse argument is identical.

Base: All instantiations fired in the first interpreter cycle of the second path will be fired in the first path.

Proof of Base: after one cycle in second path, only one rule, P_1, has fired. Assume that some instantiation $I_1 =< P_1, \theta_1 >$ fires in this cycle, but never fires in the first path. This can happen only if some production $Q \twoheadrightarrow P_1$ fires before P_1 in the first path. Since Q fires in path 1, there must be some production $R \overset{*}{\to} Q$ triggered at x, and since the system is **iio** this implies that R should have fired before P_1 in the second path, a contradiction. So all instantiations fired in the first cycle of the second path must be fired during the first path.

Induction Hypothesis: All instantiations fired in the first $k-1$ interpreter cycles of the second path are fired in the first path.

Induction: Some rule P_k fires in the k'th cycle of the second path. We shall show that all instantiations of P_k fired during this cycle are also fired in the first path. Assume that instantiation $I_k =< P_k, \theta_k >$ fires during this cycle of the second path, but never during the first path. By the induction hypothesis we know that all the instantiations causing the triggering of I_k in the second path will eventually be fired in the first path. So for I_k never to fire in the first path some instantiation $I_j =< P_j, \theta_j >$ of some production $P_j \twoheadrightarrow P_k$ must fire before P_k during the first path (thus implying, by the definition of **iio**, that $P_j \prec P_k$.)

But then how could it be that I_j did not fire in the second path as well? We shall show the only way in which this can happen, and prove that it leads to a contradiction. Once we do that, we know that I_j cannot fire in the first path without firing in the second, and thus that I_k must fire in the first path, completing the induction.

It is only possible for I_j to fire in the first path but not the second if some instantiation $I_h =< P_h, \theta_h >$ of a production $P_h \overset{*}{\to} P_j$ that caused I_j to fire in the first path did not fire before I_j in the second path. Let us consider the first such P_h to fire in the first path. How is it possible that I_h fired in the first path but not in the second? Any production $P_h \overset{*}{\to} P_j$ must have the property $P_h \prec P_k$, since we assume that $P_j \twoheadrightarrow P_k$ and the system is **iio**. So if I_h is ever to fire in the second path, it would have completed firing by now. It cannot be that some production $P_g \twoheadrightarrow P_h$ fired before P_h during the second path but not the first, since this would imply some instantiation $I_g =< P_g, \theta_g >$ must have fired before P_h (and thus before P_k) in the second path but not the first, contradicting our induction hypothesis. Thus the only way that I_h can fire in the first path but not the second is if some instantiation $I_f =< P_f, \theta_f >$ of production $P_f \to P_h$ fired before P_h in the first path but not the second. Note however, that $P_f \to P_h \overset{*}{\to} P_j$ implies that $P_f \overset{*}{\to} P_j$, and thus the assumption that I_f was fired in the first path but not the second contradicts our assumption that P_h was the first production in the first path which transitively gives-to P_j and had some instantiation not fired in the second path. Therefore it is not possible for I_h to fire in the first path, but not the second, and this implies that it is not possible for I_j to fire in the first path but not the second.

This implies that it is impossible for I_k to fire in the second path without firing in the first, and thus, by induction, we see that any instantiation fired in the second path must be fired in the first. The same argument shows that any instantiation fired in the first path must be fired in the second, and thus the set of instantiations fired in each path is identical. □

These two lemmas show that from any state x we know what instantiations will fire, and the order in which productions involved in interferences will fire. This, in essence, removes the non-determinism from the system: lemma 2 shows that all the actions taken on each side will be the same. Lemma 1 assures us that any non-determinate choices of when to take actions will not involve the shrinking of action sets of any productions. In other words, it is determined by the ordering what instantiations will ever get 'disabled' (in the sense of having parts of their conditions go from True to False during the computation) since there remains no non-determinism in the order in which a 'disabler' and a 'disablee' will get fired. The following proof puts this together to prove our main result.

Theorem 1. iio $\Longrightarrow$ ufp.

Proof. Assume the contrary. Then there exists some production system Γ which is **iio**, but has more than one fixed point out of some state, i.e. for some state $x \in S_\omega$ we have two possible computation paths, one ending in $y \in \mathbf{fp}(\Gamma)$, the other in $z \in \mathbf{fp}(\Gamma)$, $y \neq z$. Without loss of generality, assume that there is some element $k \in y$ such that $k \notin z$. Some instantiation $I =< Y_i, \theta_i >$ was the last instantiation to insert k between x and y, and by lemma 2 I must be fired at some state between x and z as well. Thus some other instantiation $J =< Z_j, \theta_j >$ must have fired after I between x and z, deleting k before z. Since lemma 2 also requires J to fire between x and y, then we must assume that J fires before I between x and z, and thus that there is no ordering between Y_i and Z_j.

We can see, however, that $Y_i \rightarrow Z_j$, since Y_i inserts k, and by the restriction we place on **delete** rules, Z_j must include k (or a variable for which k is a substitution) as part of one of its outer conjuncts. Now, there must exist productions P, Q such that $P \twoheadrightarrow Y_i$, $Q \twoheadrightarrow Z_j$, $Y_i \prec P$ and $Z_j \prec Q$. If neither existed, there would be no fixed point, since Y_i and Z_j would then continue firing alternately *ad infinitum*, inserting and deleting k over and over. If only one of P and Q existed, then it would not be possible for k to appear in y but not in z, since if Q did not exist, Z_j would always be able to fire after the last firing of Y_i, and if P did not exist, then Y_i would always be able to fire after the last firing of Z_j. Note that lemma 2 assures us that both P and Q fire on each path.

Furthermore, since $Y_i \rightarrow Z_j$, we must have the ordering $Y_i \prec Q$, by the definition of **iio**. This implies that when Q fires, neither Y_i nor Z_j is triggered. Given that Y_i and Z_j are triggered at some point in each computation path, once each of them is triggered we can see that one of them will always be triggered (alternately inserting and deleting k) until P fires, at which point Y_i will no longer be triggered, and then Z_j will delete k from working memory. Only then will Q fire (note particularly that Q will not fire before P, since one of Y_i or Z_j is always triggered before P fires). Thus when Q fires, k cannot be in working memory, implying that our assumption that $k \in y$ is impossible, and thus all fixed points must have the same set of elements, i.e. all fixed points are the same unique fixed point, so **iio** implies **ufp**. □

5. Production Memories Subject to iio

In arriving at the class of **iio** systems, we considered imposing **iio** on an arbitrary production memory. This is not possible for all sets of productions, however. This section will show exactly which production memories can have **iio** imposed upon them.

Lemma 3. *In an* **iio** *system, for productions P, Q, such that $P \neq Q$, $P \twoheadrightarrow Q$ and $Q \stackrel{*}{\rightarrow} P$, we must have $Q \prec P$.*

Proof. By the definition of **iio** we must have either $P \prec Q$ or $Q \prec P$. Assume that $P \prec Q$. Then, by the definition of **iio**, we require that $Q \prec Q$, since $Q \stackrel{*}{\rightarrow} P$. But this contradicts the definition of the production ordering relation, $\prec$, so we must have $Q \prec P$. □

Theorem 2. *We can impose* **iio** *on exactly those production memories F where*

$$\forall P, Q \in F, P \neq Q, \ (P \not\twoheadrightarrow P) \wedge (P \overset{*}{\to} Q \overset{*}{\to} P \Longrightarrow P \not\twoheadrightarrow Q)$$

that is, no self-interferences occur, and no interferences occur between productions in gives-to cycles. We say that such production memories are **iio**-able.

Proof. Since the theorem states an equivalence, the proof has two parts.

First, assume we have a system that is not **iio**-able, i.e. for some productions $P, Q \in F$, $P \neq Q$, we have that $(P \twoheadrightarrow Q) \wedge (P \overset{*}{\to} Q) \wedge (Q \overset{*}{\to} P)$, or $P \twoheadrightarrow P$. Consider the first possibility. By lemma 3, we require $Q \prec P$. But then **iio** requires that $P \prec P$, since $P \overset{*}{\to} Q$, contradicting the definition of $\prec$. Alternatively, if $P \twoheadrightarrow P$ then we have to set $P \prec P$, again a contradiction. Thus we cannot impose **iio** on systems that are not **iio**-able.

To see the converse, we shall assume a system which is **iio**-able, and construct an ordering that satisfies the conditions of **iio**. We take the dependency graph of such a system and decompose it into *strongly connected* gives-to components, each of maximium cardinality (i.e., such that two nodes are connected in a gives-to cycle iff they are in the same component). *Cross-component* gives-to edges go from productions in one component to productions in another. These gives-to connections among components form a *reduced graph*, which is guaranteed to be acyclic [1]. The acyclicity implies a partial order among the components, and we follow this by imposing the similar production ordering that if productions P and Q are in different components, $P \twoheadrightarrow Q$, and $P \overset{*}{\to} Q$, then $P \prec Q$. Similarly, if $Q \twoheadrightarrow P$ and $P \overset{*}{\to} Q$ then $P \prec Q$. All remaining unordered interferences are between productions in strong gives-to components that are not ordered in the reduced graph. We can order these components to maintain an acyclic strong-component graph, and order the interferences between these components correspondingly. This allows us to also order all resulting implied interferences consistently, since they will follow the acyclic strong component ordering, thus preserving $\prec$ as a partial order.

Such an ordering imposes **iio**, since being **iio**-able implies that interferences only occur between (not within) strongly connected gives-to components, and the partial order on components described above provides a valid production ordering which always satisfies the definition of **iio** (it does not contradict lemma 3, in particular, since the initial ordering along gives-to edges ensures that if $P \prec Q$, then $Q \overset{*}{\not\to} P$).

Thus we can impose **iio** if and only if a production memory is **iio**-able. □

The second part of this proof shows the existence of an ordering which imposes **iio** on **iio**-able production memories. But such an ordering is certainly not unique. In general, there may exist many different ways of imposing **iio** on such systems, possibly resulting in different sets of fixed points for the systems.

6. Stratified Production Systems: An Example of iio

Apt, Blair and Walker define the class of *stratified logic programs* and provide a model theory for them [2]. We present an analogous class of *stratified production systems*, which change very little of the definition of stratified logic programs; the modifications are done to bring the discussion into a production system context.[6] The differences are that we do not consider infinite states of working memory, and that we allow general first order formulas in our conditions, while the logic programs of [2] allow only clauses whose bodies form conjunctions of literals; in addition, we allow **delete** actions, while the logic programming analog, negated clause heads, are not considered in [2]. We see in this section that with regard to **ufp**, stratified production systems are in fact a special case of **iio**.

It should be noted, however, that we do not present a model theory semantics for **iio** systems that corresponds to the one that Apt *et al.* have developed for stratified programs. Their result is a powerful one,

[6] Raschid [7] has developed a similar definition of stratified production systems, but includes the concept of integrity constraints in considering systems with **delete** actions. This is not needed for our purposes in this paper, and our definitions of gives-to and interference allow a more compact definition of stratification.

because not only do they show that stratified programs have *unique supported models* (which correspond to **ufp** in our terminology) but they also prove that the unique supported model of a stratified program is the *minimal supported model* of the program. In production system terms, they prove that stratified systems have unique fixed points, and they show what the working memory state of those fixed points must be. This paper is a first step in this direction: we suspect that the unique fixed point of an **iio** system corresponds in some way to the *minimal model* of the system, but we have not established this yet.

Definition 11. *A production memory F has a* stratification *if it has a partition*

$$F = F_1 \sqcup \ldots \sqcup F_n$$

such that the following two conditions hold for $i = 1, \ldots, n$:

- *if there are two productions $P \rightarrow Q$, and Q appears in F_i, then P must be in F_j, $j \leq i$.*
- *if there are two productions $P \twoheadrightarrow Q$, and Q appears in F_i, then P must be in F_j, $j < i$.*

F_i is called the i'th stratum *of F, and we call a production memory* stratifiable *if it has at least one stratification.*

Note the implication of this for production memory dependency graphs. Stratifiable production memories are exactly those in which the dependency graph has no cycles containing an interference edge. (A formal proof of this can be found in [2].[7]) Note that stratifiable production memories are **iio**-able.

The semantics of stratifying logic programs state that rules in a stratifiable logic program are fired in a form of relational composition on $\overset{*}{\rightarrow}_{F_i}$: only rules from F_1 are fired until a state in $\mathbf{fp}(F_1)$ is reached, at which point only rules from F_2 are fired until a state in $\mathbf{fp}(F_2)$ is reached, etc., until finally a state in $\mathbf{fp}(F_n)$ is reached. In our production system model, this amounts to a production ordering which says that $P \prec Q$ iff $P \in F_i$, $Q \in F_j$, $i < j$. This is an example of **iio**, since interferences and cross-strata gives-to edges only occur from lower strata to higher.

Stratification is a stronger restriction on production memory than **iio**; although stratified production systems form a subset of **iio** systems, they are not equivalent to **iio** systems, since, as proved in the previous section, **iio** can be imposed on some systems which have cycles containing interference edges. Furthermore, stratification imposes a total order on the strata, while **iio** needs only a partial order on the strong gives-to components.

7. Modelling Production Ordering With CR-free Systems

The general model of a rule base system does not include a concept of conflict resolution; it simply is a combination of production and working memory, i.e. a CR-free system. In order to generalize the idea of production ordering, and demonstrate the applicability of our **iio** result to systems without production ordering capabilities, we develop a method for converting any production system with production ordering into a CR-free system.

The practical model of a production system is based entirely on the fixed points it reaches from a given state. In general, since the computation of a production system ends at a fixed point, two production systems may be considered computationally equivalent iff they can reach the same set of fixed points from any given state of the database[8].

Definition 12. *Two production systems Γ_1 and Γ_2 are considered* computationally equivalent *over a given universe ω if*

$$\forall x \subseteq \omega \; (x \overset{*}{\rightarrow}_{\Gamma_1} z) \wedge (z \in \mathbf{fp}(\Gamma_1)) \Longleftrightarrow (x \overset{*}{\rightarrow}_{\Gamma_2} z) \wedge (z \in \mathbf{fp}(\Gamma_2)).$$

[7] In the dependency graphs in [2], gives-to edges are called 'positive' edges, and interference edges are called 'negative' edges.

[8] An argument could be made for defining computational equivalence by way of probabilities; i.e. that two systems are computationally equivalent iff they reach the same fixed points with the same probabilities. Since we are dealing with systems that have **ufp**, these niceties become irrelevant.

Now, let us consider the specific operation of a system with production ordering. Observe that the effect of placing an ordering on two productions, $P_i \prec P_j$, is equivalent to requiring that P_j be fired only when P_i is not triggered. This can be built into the productions themselves, however, without needing to impose an external ordering, by making P_j *not be* triggered whenever P_i *is* triggered. This is stated more formally in the following:

Lemma 4. *A production system with two productions*

$$P : \delta v_{P1}, \ldots, v_{Pk}.F_P \rightarrow action_P(A_{P1}, \ldots, A_{Pr}),$$

$$Q : \delta v_{Q_1}, \ldots, v_{Q_l}.F_Q \rightarrow action_Q(A_{Q_1}, \ldots, A_{Q_s})$$

such that $P \prec Q$, is computationally equivalent to the same system with Q replaced by

$$Q' : \delta v_{Q_1}, \ldots, v_{Q_l}.F_{Q'} \rightarrow action_Q(A_{Q_1}, \ldots, A_{Q_s})$$

where

$$F_{Q'} \equiv \begin{cases} \forall v_{P1}, \ldots, \forall v_{Pk}(F_Q \wedge (\neg F_P \vee (A_{P1} \wedge \ldots \wedge A_{Pr}))) & \textit{if } action_P = \textbf{insert} \\ \forall v_{P1}, \ldots, \forall v_{Pk}(F_Q \wedge (\neg F_P \vee (\neg A_{P1} \wedge \ldots \wedge \neg A_{Pr}))) & \textit{if } action_P = \textbf{delete} \end{cases}$$

and the production ordering is altered so that there is no order between P and Q.

Proof. Consider any $x \in S_\omega$, a state of the database. Two cases are considered:

1. If F_P is not satisifed over x, then $F_{Q'}$ will be satisfied over x iff F_Q is satisfied over x.
2. If F_P is satisfied over x by some substitution θ then $F_{Q'}$ is satisfied iff all the actions of instantiation (P, θ) have already been carried out for all such θ.

Note that $P \neq Q$, by the definition of $\prec$. Now, in the first case above, P is never triggered, so Q' is triggered whenever F_Q is satisfied. In the second case, even if F_P is satisfied by some θ, P is not triggered (i.e. P has no *non-trivial* instantiation) if all of the action elements of (P, θ) have been carried out, and in this situation Q' is triggered if F_Q is satisfied. Thus Q' is fired only if P is not triggered and F_Q is satisfied, and the effect of firing Q' is identical to the effect of firing Q. Therefore at any state $x \in S_\omega$, P and Q' behave, without a production ordering, the same way that P and Q behave under the condition that $P \prec Q$, and thus the new system is computationally equivalent to the original. □

The natural result of the above lemma is that by applying it pairwise on all the productions of a system, we can model the system as a new CR-free system, and have the same computational outcome.

Theorem 3. *For each production system with production ordering there exists a computationally equivalent CR-free production system.*

Proof. By construction. We take an arbitrary production system Γ with production memory F that has a production ordering, and we build a new set of productions forming a new system Γ' with production memory F' which is CR-free, but identical to Γ in its actions, as described by the following algorithm. As a convention, we consider productions P and Q to be defined as in the preceding lemma, with arbitrary δ-prefixes, formulas and actions.

let the working memory of Γ' equal that of Γ
let $F' = \emptyset$
let Γ' have no conflict resolution
for each production P in F **do**
  if P is not in F', add it, but call it P'
  for each production Q such that $P \prec Q$ in F **do**

if Q' is already in F' **then**

let $F_{Q'}$ be $\begin{cases} \forall v_{P1}, \ldots, \forall v_{Pk}(F_{Q'} \wedge (\neg F_P \vee (A_{P1} \wedge \ldots \wedge A_{Pr}))) & \text{if } action_P = \textbf{insert} \\ \forall v_{P1}, \ldots, \forall v_{Pk}(F_Q \wedge (\neg F_P \vee (\neg A_{P1} \wedge \ldots \wedge \neg A_{Pr}))) & \text{if } action_P = \textbf{delete} \end{cases}$

else add a new production, Q', to F':

$Q' : \delta v_{Q_1}, \ldots, v_{Q_l}.F_{Q'} \rightarrow action_Q(A_{Q_1}, \ldots, A_{Q_s})$ and

$F_{Q'} \equiv \begin{cases} \forall v_{P1}, \ldots, \forall v_{Pk}(F_Q \wedge (\neg F_P \vee (A_{P1} \wedge \ldots \wedge A_{Pr}))) & \text{if } action_P = \textbf{insert} \\ \forall v_{P1}, \ldots, \forall v_{Pk}(F_Q \wedge (\neg F_P \vee (\neg A_{P1} \wedge \ldots \wedge \neg A_{Pr}))) & \text{if } action_P = \textbf{delete} \end{cases}$

Note that for any production Q' in F', it has the same δ-prefix as Q in F, and also the same right-hand-side, and thus remains consistent in that sense. Furthermore, note that whenever we have two productions P, Q such that $P \prec Q$, Q' is modified to be triggered only when P is not triggered. The result of the algorithm, then, is that for any production Q in F, there is a production Q' in F' that has the same actions as Q, but $F_{Q'}$ is changed so that Q' is triggered (following lemma 4) iff:

1. F_Q is satisfied and
2. no F_P is triggered such that $P \prec Q$ in the original system.

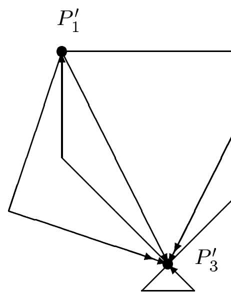
Thus this algorithm produces a production system which, with only an Arbitrary Decision conflict resolution scheme, acts identically at any state $x \in S_\omega$ to a system with production ordering. This proves that the two systems are computationally equivalent. □

This theorem frees us from considering systems with production ordering, since we may instead consider their CR-free equivalents. It should be noted however, that this theorem does not rid production ordering of its utility, for the equivalent systems, while easier to analyze in general, are considerably more complex to build, and it is more difficult to get an intuitive grasp of their specific workings.

Example 7. Consider our example system, but with the production ordering $\{(P_2 \prec P_3)\}$. Such a system has a CR-free equivalent with the following set of productions:

$$P_1' : \delta v_1, v_2.\neg(\text{Jerry}, \text{MOUSE}) \wedge (\text{CHEESE}) \rightarrow \textbf{insert}(\text{Mickey}, \text{MOUSE})$$
$$P_2' : \delta v_1.(v_1, \text{MOUSE}) \rightarrow \textbf{insert}(\text{Fido}, \text{DOG})$$
$$P_3' : \delta v_1, v_2.\forall v_3(\neg(\text{Fido, DOG}) \wedge (v_1, \text{MOUSE}) \wedge (v_2, \text{CAT}) \wedge (\neg(v_3, \text{MOUSE}) \vee (\text{Fido}, \text{DOG})))$$
$$\rightarrow \textbf{delete}((v_1, \text{MOUSE}))$$

The corresponding dependency graph is shown below:



We can now see that an implied interference in an ordered system becomes a real interference in the equivalent CR-free system, as demonstrated in the example above. This results from the fact that if we set $P_2 \prec P_3$, then F_{P_3}' contains $\neg F_{P_2}$, and thus firing $P_1' \rightarrow P_2'$ can disable all instantiations of P_3' by satisfying F_{P_2}.

8. Summary and Directions for Further Investigation

The development of **iio** systems and the proof of their determinism has interesting practical ramifications for production systems, and for systems which couple rule bases and an interpreter to traditional databases.

The **iio** systems extend the set of systems with **ufp** past stratification, and therefore expand the number of systems that can be proven to be deterministic.

This result would be greatly enhanced by intelligent algorithms for discovering gives-to and interference relations among productions, since this would allow systems to suggest orderings which impose **iio**, and also examine existing orderings to see if they conform to the constraints of **iio**. Zhou and Hsu develop a conservative algorithm for quickly finding a superset of the interference relations among productions [13], but do not suggest an efficient algorithm for exactly computing all gives-to and interference relations. Such an algorithm would provide a significant tool to aid production system developers to build deterministic systems, and could be used to help better understand existing systems, deterministic or otherwise.

The applicability of **iio** to production systems with more complicated rule semantics ([10], etc.) seems promising, since the definitions of gives-to and interferes drive the proofs, and are not dependent on the rule semantics used here. However the problem of discovering the gives-to and interference relations becomes more complicated as the rule semantics become more complicated, and the practical utility of the results here depends heavily on the difficulty of discovering these relations.

Our **iio** $\Longrightarrow$ **ufp** result should hold for systems that use somewhat different firing policies than the rule-at-a-time scheme considered here; a variety of instantiation-based firing schemes subject to production ordering would not change our results. This was not undertaken here for the sake of brevity.

Our other result, a method for converting a production ordering system to a computationally equivalent CR-free system, is a tool which is interesting in its own right, since it allows us to view the **iio** systems as a class of general CR-free systems, which correspond quite closely to logic programs. This has implications for the plausability of developing a model theory for the CR-free equivalents of **iio** systems, along the lines of that which exists for stratified logic programs [2]. The CR-free equivalents of **iio** systems should have unique supported models, may have least supported models, and promise to be an interesting new class of logic programs.

In another direction, this proof points to the possibility of abstracting further conflict resolution schemes as CR-free systems. *Specificity ordering* [5] is a production ordering scheme that makes $P \prec Q$ if F_P is satisfied in a subset of the states where F_Q is satisfied. It would be interesting to examine how specificity ordering relates to **iio**, and to explore the properties of the CR-free equivalents of specificity ordered systems.

Acknowledgements.

Thanks to Jennifer Widom for extensive feedback on form and content, and to Yuli Zhou for helpful suggestions and clarifications.

References.

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[2] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89 – 148, Morgan Kaufmann Publishers, 1988.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison-Wesley Publishing Company, 1985.

[4] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley Publishing Company, 1986.

[5] J. McDermott and C. Forgy. Production system conflict resolution strategies. In [9], pages 177 – 199.

[6] John McDermott. R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19(1), 1982.

[7] Louiqa Raschid. Maintaining consistency in a stratified production system program. In *Proceedings of the AAAI National Conference on Artifical Intelligence*, 1990.

[8] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, second edition, 1982.

[9] D. A. Waterman and Frederick Hayes-Roth, editors. *Pattern-Directed Inference Systems.* Academic Press, Inc., 1978.

[10] J. Widom and S. J. Finkelstein. *A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems.* IBM Research Report, IBM Almaden Research Center, 1989. A version of this paper appears in SIGMOD '90.

[11] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, second edition, 1984.

[12] Yuli Zhou and Meichun Hsu. Rule bases with the unique fixed point property. February 1990. Submitted for publication.

[13] Yuli Zhou and Meichun Hsu. A theory for rule triggering systems. In *Conference on Extending Database Technology (EDBT '90)*, March 1990.