A Three-Valued Semantics for Deductive Databases and Logic Programs*

JIA-HUAI YOU AND LI YAN YUAN

Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2H1

Received March 20, 1990; revised June 9, 1993

This paper proposes two principles, justifiability and minimal undefinedness, for a three-valued model-theoretic approach to semantics of logic programs and deductive databases (also called disjunctive logic programs). The former is intimately related to the concept of labeling-based justification in Doyle's truth maintenance system while the latter requires the use of the truth value undefined only when it is necessary. We examine the question why and in what circumstances the undefined is needed under these two principles. We show that these two principles yield a declarative semantics for deductive databases and logic programs, which is called the regular model semantics. Program properties in this semantics are analyzed and results obtained concerning the relationship among regular, stable, and well-founded semantics, which show that the regular model semantics is a natural extension of the latter two semantics. © 1994 Academic Press, Inc.

1. INTRODUCTION

One of the challenging problems in the field of deductive databases and logic programming has been a declarative semantics for an arbitrary set of clauses whose body may contain negative literals and whose head is a disjunction of atoms. Following [19], such a first-order theory is called a deductive database. It is also called a disjunctive program in the literature. As the special case when the head of each clause is a singleton, it is called a logic program.

An early approach to logic program semantics has been Clark's predicate completion whose main advantage is a formalism that completely stays within the traditional first-order logic [3]. One of the problems with this approach is that a completed program may not always possess a model, which has been addressed by Fitting using a three-valued approach [8] (also see [15, 28]). Other unintuitive features of this approach have also been pointed out in the literature [23].

To overcome these problems, an important class of logic programs, called stratified logic programs, that disallow recursion through negation, has been

* This is a substantially improved and extended version of an extended abstract that appeared in the "Proceedings of the 9th ACM PODS, 1990" [30]. Work supported by the Natural Sciences and Engineering Research Council of Canada.

0022-0000/94 $6.00
Copyright © 1994 by Academic Press, Inc.
All rights of reproduction in any form reserved.
identified and its semantics defined [1, 9, 20]. This approach has been extended to a wider class of programs called \textit{locally stratified} programs [19].

It was soon realized that non-locally stratified programs make practical sense and logic program semantics should be extended to allow a wider class of programs [12]. The two dominant approaches so far have been a two-valued formalism—the stable model semantics [12], and a three-valued one—the well-founded model semantics [10].

The stable model semantics is based on an argument from autoepistemic logic that an intended model of a program should be a possible set of beliefs that a rational agent might hold. A drawback of this semantics is that a program may not always possess a stable model. On the other hand, the well-founded approach only computes the minimal amount of information that can be definitely determined by a well-founded computation; those that cannot be determined will then be treated as the \textit{undefined}. Thus, every program possesses at least one three-valued model. Przymusinski provided an elegant fixpoint definition of the well-founded semantics in a formal three-valued logic and extended the approach to some of the major nonmonotonic reasoning formalisms [22, 24]. We argue that this formalization may result in loss of disjunctive information embedded in a program. This is particularly true when it is applied to deductive databases and general nonmonotonic reasoning. As a matter of fact, the problem is rather deeply rooted in a reasoning pattern that uses clauses as rules, where the orientation of clauses is of vital importance, and in the three-valued formalization itself. As a result, the notion of the undefined is rather mysterious; there does not appear to exist a logical explanation of the undefined, except that the undefined are those that cannot be well-foundedly computed.

This paper provides new insights into the semantics issues of logic programs and deductive databases. Especially, the notion of the undefined in three-valued logic is examined. Our starting point is, if all the reasons for the need and elegance of three-valued formalization are because of certain difficulties in using two truth values only, then the undefined should be used only when it is necessary. We call this principle \textit{minimal undefinedness}. Przymusinski's three-valued formalization does not meet this principle; i.e., a three-valued (minimal) model therein can be "over-defined." This leads to the loss of disjunctive information in its semantics definition.

The principle of minimal undefinedness, however, should be imposed on top of another principle, called \textit{justifiability}, which is intimately related to the concept of labeling-based justification in Doyle's truth maintenance system [4]. Based on these two principles, we define a new semantics of logic programs and deductive databases, which we call \textit{the regular model semantics}.\footnote{During the review of this paper, an abduction-based semantics for logic programs, called the \textit{preferential semantics}, was proposed by Dung [6]. It can be shown that, for logic programs, the preferential semantics coincides with the regular model semantics. The latter, however, is also defined for all deductive databases, or disjunctive programs.} This semantics definition can
be viewed as taking the best of both the well-founded semantics and the stable model semantics. Consequently, every program possesses at least one regular model and disjunctive information can be restored from the existence of multiple regular models. More importantly, we show that an undefined atom in the well-founded model of a program may denote some useful information: disjunctive and factoring information, both of which have been accommodated in the regular model semantics. The atoms that truly belong to the undefined are those that must not be assigned true nor false according to the two principles. By the very nature of these atoms, they are called "difficult-to-be-assigned." This provides an interesting interpretation of stable models in terms of regular models and explains why and in what circumstances a program fails to possess a stable model: stable models are exactly two-valued regular models; a nontrivial three-valued regular model (with a non-empty set of the undefined) corresponds to a "disappeared stable model" because of the "difficult-to-be-assigned" atoms; and when every regular model is nontrivially three-valued, a program fails to possess a stable model.

We show a syntactic sufficient condition and a dynamic sufficient condition, under which all regular models of a program are two-valued and coincide with its stable models. We show that programs violating the dynamic condition are often unclear, if not completely ambiguous, in their intended interpretation. This raises the question of whether the undefined is truly needed for practically useful programs.

The handling of disjunctive information can be computationally expensive. We believe that any reasonable semantics that intends to handle disjunctive information and that lands itself in the notion of minimizing positive conclusions cannot escape this fate. However, we have shown in a separate paper [31] that one can use circumscription to compute sentences true in every stable model and, therefore, in every two-valued regular model. A partial proof theory is thus shown to exist by the facts that there is a sufficient condition to guarantee any regular model of a deductive database to be two-valued and that there exist query-answering algorithms for circumscriptive theories, such as MILO-resolution [21]. In addition, because the well-founded model is contained in every regular model, any query proved by a proof procedure for the former is a correct answer for the latter. Even without a complete proof theory for the regular model semantics at the current stage, we believe that the results presented here regarding the semantic issues of deductive databases and logic programs are important in their own right.

The paper is organized as follows. In the next section, we will provide a review over three-valued logic as well as the well-founded and stable model semantics. Section 3 will provide motivating examples for the new semantics to be proposed in this paper. The two principles will be defined in Section 4 with the main results of this paper given in Section 5.

As some proofs in this paper require additional background, which is a separated and dedicated matter and which is of interest in its own right, we will put this background material and these proofs in the Appendix.
2. THREE-VALUED FORMALISM AND LOGIC PROGRAM SEMANTICS

2.1. Basic Definitions

We assume the well-known concepts and notations of traditional two-valued logic of a first-order language $L$.

We call an atom a positive literal and an atom with the negation sign in front a negative literal. By a deductive database (DB), we mean a finite set of universally quantified clauses of the form: $A_1, ..., A_m \leftarrow L_1, ..., L_n$, where $n \geq 0$, $m \geq 1$, and the $L_i$ are literals and the $A_i$ are atoms. As usual, the body of a clause denotes a conjunction of literals and the head a disjunction of atoms. A deductive database is said to be positive when the body of each clause therein contains positive literals only. We also call clauses oriented clauses or rules to emphasize the importance of the orientation of these clauses in a deductive database.

A clause denotes a (finite or infinite) set of ground clauses that are instantiated by the terms from the Herbrand universe of the language. We therefore assume ground deductive databases and logic programs if not otherwise said. A deductive database is called a logic program, or just a program, if the head of each clause is an atom. Given a program and an atom, by a definition clause of the atom we mean a clause with the atom being its head. Throughout this paper, we consider only Herbrand models. The Herbrand base of the language $L$ will be denoted by $H_L$.

In a three-valued interpretation $I$, a third truth value, called undefined, can be given to well-formed formulas of $L$. We denote by $\langle T; F \rangle$, or sometimes by $\langle T; F; U \rangle$, a three-valued interpretation, where $T$ contains all ground atoms true in $I$, $F$ contains all ground atoms false in $I$, and $U = H_L - (T \cup F)$ the remaining ground atoms of $H_L$ undefined in $I$. We denote by $t$, $f$, and $u$, respectively, the truth values true, false, and undefined. Here we are only interested in consistent interpretations where $T \cap F = \emptyset$. If $A$ is a ground atom from $H_L$, we then denote by $\text{val}_I(A)$ the truth value of the atom in the interpretation $I$.

The intersection of interpretations $I_i = \langle T_i; F_i \rangle$, for $i \in \Omega$, is defined as $I = \langle \bigcap_{i \in \Omega} T_i; \bigcap_{i \in \Omega} F_i \rangle$. It is easy to verify that the intersection of consistent interpretations is a consistent interpretation.

An interpretation $I = \langle T; F \rangle$ is an extension of another interpretation $I' = \langle T'; F' \rangle$, denoted by $I' \subseteq I$ if $T' \subseteq T$ and $F' \subseteq F$.

Let $S$ and $Q$ be sentences of $L$. To determine the truth value of a more complex sentence, the following evaluation rules are used:

The ordering of the truth values is given by $f < u < t$;

\[
\text{val}_I(\neg S) = \neg \text{val}_I(S), \text{ where } \neg t = f, \neg f = t, \text{ and } \neg u = u;
\]

\[
\text{val}_I(S \lor Q) = \max\{\text{val}_I(S), \text{val}_I(Q)\}, \text{ where the maximum of an empty set is } f;
\]

\[
\text{val}_I(S \land Q) = \min\{\text{val}_I(S), \text{val}_I(Q)\}, \text{ where the minimum of an empty set is } t;
\]
val_I(∃xS(x)) = \max\{val_I(S(a)) \mid a \in H_L\};
val_I(∀xS(x)) = \min\{val_I(S(a)) \mid a \in H_L\};
val_I(Q ← S) = t \quad \text{if} \quad val_I(S) \leq val_I(Q)
= f \quad \text{otherwise.}

Note that a sentence in the implication form evaluates to t if both sides are undefined. This is different from another implication symbol ⊨ of three-valued logic, which is defined as: Q ⊨ S ≡ Q ∨ ¬S. Note that Q ⊨ S is not logically equivalent to Q ∨ ¬S. We will illustrate in Section 3 that much of the controversy in logic program semantics is centered around this discrepancy.

We will use Π to denote a DB if not otherwise said. A three-valued interpretation I is a model of Π if every clause in Π evaluates to true in I. A three-valued interpretation \langle T; F \rangle is said to be two-valued if T ∪ F = H_L. A two-valued interpretation is also denoted by a set with the understanding that the atoms in the set are assigned to true and the rest to false. Since two-valued interpretations are a special class of three-valued ones, it is understood that the word model refers to three-valued in the general case; we use two-valued or three-valued for emphasis when there is a need. When T ∪ F ≠ H_L, \langle T; F \rangle is said to be a nontrivial three-valued model.

Finally, a model M = \langle T; F \rangle of Π is less than or equal to a model M' = \langle T'; F' \rangle of Π, denoted by M ≤ M', if T ⊆ T' and F' ⊆ F. As usual, we will use M < M' to mean M is less than M'. A model is said to be minimal if no other model is less than it.

Note that if both M and M' are two-valued, that M < M' simply means M ⊆ M', and the notion of minimal model coincides with the standard one in two-valued logic.

For more background on three-valued logic and their applications in logic programming with negation, see [8, 14, 15, 28]. Particularly, Fitting and Kunen [8, 15] used a different three-valued logic to justify their semantics in accordance with completed programs.

2.2. Well-Founded and Stable Model Semantics

2.2.1. Fixpoint Computation of Well-Founded Model

We give Przymusinski's version of fixpoint computation of well-founded models [24]. Other equivalent definitions can be found in [2, 11].

The well-founded model of a logic program is constructed by using two monotonic operators T_I and P_I which compute, based on the currently known facts in I, the new positive facts and negative facts, respectively. This process is iterated by another monotonic operator I until no more new facts, either positive or negative, can be generated.
Let $T$ and $F$ be sets of ground atoms and $\Pi$ be a program. Define:

$$T_i(T) = \{A \mid \text{there is a clause } A \leftarrow L_1, \ldots, L_m \text{ in } \Pi \text{ such that } L_i \text{ is either true in } I \text{ or } L_i \in T \text{ for all } 1 \leq i \leq m\};$$

$$\Phi_i(F) = \{A \mid \text{for each clause } A \leftarrow L_1, \ldots, L_m \text{ in } \Pi, \text{ there exist } L_i, 1 \leq i \leq m, \text{ such that } L_i \text{ is either false in } I \text{ or } L_i \in F\}.$$

The operator $T_i$ always starts from the empty set and new positive facts are then added, while the operator $\Phi_i$ starts from the entire Herbrand base and repeatedly tries to eliminate the atoms that cannot be determined false for the time being.

Let $I = \langle T; F \rangle$ be an interpretation. Define:

$$T^{\to}_t = \emptyset \text{ and } F^{\to}_t = H_L,$$

$$T^{\to}_{i+1} = T_i(T^{\to}_t) \text{ and } F^{\leftarrow}_{i+1} = \Phi_i(F^{\leftarrow}_t),$$

$$T_t = \bigcup_{n < \omega} T^{\to}_t \text{ and } F_t = \bigcap_{n < \omega} F^{\leftarrow}_t.$$

The least fixpoint of the operator $T_t$ (resp. $\Phi_t$) is then $T_t$ (resp. $F_t$).

Define an operator $I$, where

$$I(I) = I \cup \langle T_t; F_t \rangle$$

$$I^{\to} = \langle \emptyset; \emptyset \rangle \text{ and } I^{\leftarrow+n+1} = I(I^{\leftarrow n}).$$

The well-founded model of a given program $\Pi$ is defined as the least fixpoint of the operator $I$.

2.2.2. Stable Model Semantics

The stable model semantics uses an argument from autoepistemic logic that an intended model of a program should be a possible set of beliefs that a rational agent might hold. The belief set of a rational agent can be thought of as being established in the following way: for any subset $M$ of the Herbrand base, let $\Pi_M$ be the program obtained from $\Pi$ by deleting

(i) each rule that has a negative literal $\neg B$ in its body with $B \in M$, and  
(ii) all negative literals in the bodies of the remaining rules.

Let $M'$ be the least model of the modified program (which always exists since $\Pi_M$ is a positive program [7]). If $M' = M$, $M$ is said to be a stable model of $\Pi$.

3. Motivating Examples

Example 3.1. Let $\Pi$ consist of a single clause $q \leftarrow \neg p$. There are three minimal models:

$$M_1 = \langle \{p\}; \{q\} \rangle$$

$$M_2 = \langle \{q\}; \{p\} \rangle$$

$$M_3 = \langle \emptyset; \emptyset \rangle.$$
Przymusinski's three-valued parallel circumscription is defined using all three-valued minimal models [22], while two-valued parallel circumscription only considers the first two minimal models. It is apparent that the latter yields more information than the former. This huge semantic difference makes three-valued logic sometimes unintuitive.

This semantic disparity would not have occurred should we have used $\iff$ instead of $\leftrightarrow$. In that case, the only minimal models in the preceding example are $M_1$ and $M_2$. The problem with $\iff$ is twofold. First, we note that all minimal models of a DB using $\iff$ are two-valued (see Theorem 4.1). This fact seems to suggest that a three-valued approach is superficial. However, to a certain extent $\iff$ is inadequate to represent clauses as rules.

It has been generally agreed that an arrangement of clauses in a logic program into the implication form bears certain intended meanings. For example, the model $M_2$ in Example 3.1 serves as the intended meaning in the perfect model semantics because of the priority relation that results from the orientation of the clause. Under this assumption, a set of clauses is no longer treated as a first-order formula semantically, but rather, as a set of rules that denotes a certain first-order theory whose consequences can be derived from these rules.

**Example 3.2.** Let $\Pi$ be $\{a \iff \neg a\}$. This example has been repeatedly used in the literature, but we still feel a need of clarification. The program has a unique three-valued minimal model $M = \langle \emptyset; \emptyset \rangle$. If we replace $\iff$ by $\leftrightarrow$, the only minimal model will be $M' = \langle \{a\}; \emptyset \rangle$. These two interpretations represent two different approaches. The first one is based on the reasoning pattern where a rational reasoning agent would not "derive" a positive fact by a false condition (see, for example, [10, 12, 24, 29]). Thus $a$ cannot be true. Since $a$ being false simply makes the clause unsatisfied, atom $a$ becomes "difficult-to-be-assigned." The second approach relies on an extended priority relation [18] from which the conclusion that $a$ is true can be drawn. Indeed, in two-valued logic, $a \iff \neg a$ is logically equivalent to $a$.

It can be argued that the first approach appears to be more intuitive in its reasoning pattern: program clauses should be treated as rules (which should also be satisfied). As a rule, a clause can be applied only if its body is satisfied. Under this reasoning pattern, any positive conclusion should be able to be demonstrated by following the orientation of the given rules. This approach represents a more dramatic departure from the concepts of classic logic.

A nontrivial result of this paper is that the phenomenon that *an atom negatively depends upon itself* is the only situation where the undefined is needed in the first approach. The loss of disjunctive information in the well-founded semantics can be seen from the following example.
Example 3.3. Let $\Pi$ consist of

\begin{align*}
in\text{-}class(joe, cs-100) \\
professor(x) \leftarrow in\text{-}class(x, cs-100), \neg student(x) \\
student(x) \leftarrow in\text{-}class(x, cs-100), \neg professor(x).
\end{align*}

The most plausible meaning of the program is that of an indefinite situation: we have no information about whether joe is a professor or a student when we see him in class cs-100; but we know he must be either a student or a professor. Accordingly, this disjunctive information should be obtainable as a consequence of the program.

Unfortunately, the three-valued formalization fails to capture this intuition. To simplify our discussion, let us consider only the instance of the program above by replacing $x$ in all the clauses by $joe$. There then exist three 3-valued minimal models for this ground program:

\begin{align*}
M_1 &= \{in\text{-}class(joe, cs100), student(joe)\}; \{professor(joe)\} \\
M_2 &= \{in\text{-}class(joe, cs100), professor(joe)\}; \{student(joe)\} \\
M_3 &= \{in\text{-}class(joe, cs100)\}; \emptyset.
\end{align*}

Under the well-founded semantics, the meaning of the program is denoted solely by the model $M_3$. Apparently, what should have been treated as disjunctive information has been treated as undefined. This treatment is inadequate for deductive databases where disjunctive information is of vital importance.

It is worth noting that $M_3$ is "over-undefined" in that $student(joe)$ and $professor(joe)$ could have been assigned true or false. This phenomenon of over-undefinedness is caused by the definition of $\leftarrow$: an implication is true if both sides are undefined. We should also mention that $M_3$ is the intersection of $M_1$ and $M_2$; this reveals that disjunctive information is implicitly embedded in the undefined.

4. Two Principles of Nonmonotonic Reasoning by Oriented Clauses

In this section we define two principles of nonmonotonic reasoning by oriented clauses. Our starting position for the use of the undefined is that if the rationals behind the adoption of the three-valued formalization stem from the difficulties in relying on the two truth values only, such as the situation in Example 3.2, then the undefined should be assigned only to those atoms that indeed cannot be assigned otherwise. A desired model with less undefined atoms should then overwrite those with more undefined. Precisely,

Definition 4.1. Let $\Sigma$ be a set of models of a DB $\Pi$. A model $M_1 = \langle T_1; F_1; U_1 \rangle$ in $\Sigma$ is said to be less undefined than another model $M_2 = \langle T_2; F_2; U_2 \rangle$ in $\Sigma$, denoted $M_1 <_{\text{undefined}} M_2$, iff $U_1 \subseteq U_2$, $T_1 \models T_2$, and $F_1 \supseteq F_2$. 
A model $M$ from $\Sigma$ is said to be minimally undefined in $\Sigma$ iff there does not exist a model $M'$ in $\Sigma$ such that $M' <_{\text{undef}} M$.

Note that the given set $\Sigma$ of models in the definition is left unspecified for the general case. It will be determined in this paper by another desirable property—justifiability, which will be introduced shortly.

**The Principle of Minimal Undefinedness.** The only interesting models in a set of models are those that are minimally undefined in the set.

This principle requires that the truth value undefined be used as conservatively as possible.

**Theorem 4.1.** (i) Let $\Pi$ be a DB with all occurrences of $\leftarrow$ replaced by $\Leftarrow$. Let $\Sigma$ be the set of all minimal models of $\Pi$. Then, every model $M$ in $\Sigma$ is two-valued and therefore trivially minimally undefined in $\Sigma$.

(ii) There exists at least one DB that possesses a nontrivial, minimally undefined model in the set of all its minimal models.

**Proof.** (i) By definition, each clause with implication $\Leftarrow$ is logically equivalent to a disjunction of literals. Such a set of clauses is obviously satisfiable. Let $M = \langle T; F; U \rangle$ be any minimal model of $\Pi$ where $U \neq \emptyset$. Since $T$, $F$, and $U$ are pairwise disjoint, $M' = \langle T; F \cup U; \emptyset \rangle$ still satisfies every clause. But we have $M' < M$. This contradicts the minimality assumption of $M$.

(ii) See Example 3.2.

The theorem means that if the implication symbol $\Leftarrow$ is adequate for deductive databases and logic programs, the truth value undefined would never be needed. However, the employment of $\Leftarrow$ would make it difficult to enforce a certain desirable reasoning pattern which is embodied in the second principle regarding justifiability of derived facts. We first define justifiability for logic programs.

**Definition 4.2.** Let $\Pi$ be a logic program. A three-valued model $M = \langle T; F \rangle$ of $\Pi$ is said to be justifiable iff every atom $Q$ in $T$ is justified in that there exists a clause

$$Q \leftarrow C_1, \ldots, C_k, \neg B_1, \ldots, \neg B_m$$

in $\Pi$ such that $\{B_1, \ldots, B_m\} \subseteq F$ and $C_1, \ldots, C_k, k \geq 0$, have already been justified.

It is easy to see that for any two-valued model $M$, $M$ is a stable model of $\Pi$ iff $M$ is a justifiable model of $\Pi$. In fact, the Gelfond–Lifschitz's transformation can be extended to three-valued-models for deductive databases in a similar way.

Let $M = \langle T; F \rangle$ be a three-valued model of a DB $\Pi$. $\Pi$ can be transformed to a positive DB, denoted by $\Pi_M$, as follows: (i) remove any $\neg B$ from the body of a clause if $B \in F$, and (ii) remove the clauses with any negative literals in their bodies.
A positive DB has at least one 2-valued minimal model. Thus we define

**Definition 4.3.** Let $\Pi$ be a DB. A three-valued model $M = (T; F)$ of $\Pi$ is said to be justifiable iff $T$ is a two-valued minimal model of $\Pi_M$.

**Proposition 4.2.** For logic programs, Definition 4.3 is equivalent to Definition 4.2.

**Proof.** Let $M = (T; F)$ be a model of a logic program $\Pi$. Suppose $T$ is a minimal model of $\Pi_M$. Since $\Pi_M$ is a positive program, $T$ is the least model of $\Pi_M$. Thus every atom in $T$ is justifiable by Definition 4.2. For the reverse, suppose that every atom in $T$ is justifiable by Definition 4.2. Let $P$ be the set of the clauses in $\Pi$ that are involved in justifying any atoms in $T$, and let $P'$ be the set of the clauses in $P$ with negative literals removed by the transformation. Clearly, $P' \subseteq \Pi_M$ and $T$ is the least model of the positive program $P'$. Since for any clause $r \in \Pi_M - P'$, $r$ is not involved in justifying any atom in $T$, the body of $r$ is false in $T$. Therefore, $T$ is also the least model of $\Pi_M$.

**The Principle of Justifiability.** For oriented clauses, the only interesting models are those that are justifiable.

The principle of justifiability defined here is essentially the same concept as labeling-based justification in Doyle's truth maintenance system [4]. Justifiability requires that any positive conclusion be able to be demonstrated by the reasoning that follows the orientation of the rules, because this orientation reflects one's intuition about the way the reasoning should be performed. An atom can be assumed false if it leads to a consistent argument. It was shown in [29] that the unintuitive extension in the Hanks-McDermott shooting problem [13] (also see [17]) resulted from a violation of this principle.

Almost all logic program semantics obey the principle of justifiability. We summarize these results in the following proposition whose proof is straightforward.

**Proposition 4.3.** Let $\Pi$ be a DB. A two-valued model $M$ of $\Pi$ is justifiable iff

(a) it is the least model of $\Pi$ if $\Pi$ is a positive program;
(b) it is a perfect model of $\Pi$ in the case that $\Pi$ is a stratified deductive database [19]; and
(c) it is a stable model of $\Pi$ if $\Pi$ is a logic program.

Furthermore, if $\Pi$ is a logic program, then the well-founded model of $\Pi$ is justifiable.

Although a two-valued justifiable model of a program is automatically a minimal one, a three-valued justifiable model may not be a three-valued minimal model. As a proof of this claim, consider $P = \{a \leftarrow b\}$, of which $M = (\emptyset, \{b\})$ is a justifiable model but not a three-valued minimal model.

It now becomes clear that had we used the implication $\Leftarrow$ in logic programs, justifiability would have forced certain programs, such as that in Example 3.2, to lose all their minimal models. This is the key reason why some logic programs fail to have a stable model.
5. THE REGULAR MODEL SEMANTICS

In this section we define the regular model semantics for deductive databases and logic programs.

**Definition 5.1.** Let $\Pi$ be a DB and let $\Sigma$ be the set of all justifiable models of $\Pi$. A three-valued model of $\Pi$ is said to be regular if it is minimally undefined in $\Sigma$. The regular model semantics of $\Pi$ is defined by all the regular models of $\Pi$.

There are two reasoning modes based on regular models: brave reasoning with respect to some particular regular model or skeptical reasoning with respect to all regular models.

**Example 5.1.** The unique regular model in Example 3.1 is $M_2$. The unique model in Example 3.2 is regular. So are the first two minimal models in Example 3.3.

**Example 5.2 (Przymusinski 1989 [24]).** Consider the program:

\[
\begin{align*}
b & \leftarrow \neg a \\
c & \leftarrow \neg b \\
c & \leftarrow a, \neg p \\
p & \leftarrow \neg q \\
q & \leftarrow b, \neg p.
\end{align*}
\]

The program's well-founded model is $\langle \{b\}; \{a, c\} \rangle$. There are two regular models: $M_1 = \langle \{b, p\}; \{a, c, q\} \rangle$ and $M_2 = \langle \{b, q\}; \{a, c, p\} \rangle$, both of which are two-valued and extensions of the well-founded model. In addition, their intersection coincides with the well-founded model.

**Proposition 5.1.** Every DB has at least one regular model.

**Proof:** We first show that every DB $\Pi$ has at least one justifiable model. Let $\Pi'$ be the set of the positive clauses in $\Pi$ and let $T$ be a two-valued minimal model of $\Pi'$. The three-valued interpretation $\langle T; \emptyset \rangle$ is clearly justifiable. Thus we only need to show that $\langle T; \emptyset \rangle$ is a model of $\Pi$. For any clause $r$ in $\Pi - \Pi'$, the body of $r$ is either false or undefined in $\langle T; \emptyset \rangle$ and the head is either true or undefined in $\langle T; \emptyset \rangle$. In either case, the clause $r$ is satisfied. Since $<_{\text{undef}}$ is a quasi-order on the set of justifiable models of the program over the domain $H_L$, there always exists a regular model. 

In the definition of regular model, there is not explicit requirement that a regular model be a three-valued minimal model. However,

**Proposition 5.2.** A regular model of a DB $\Pi$ is a three-valued minimal model of $\Pi$. 
Proof. Let \( M_R = \langle T_R; F_R \rangle \) be a regular model of \( \Pi \). Assume there exists a model \( M = \langle T; F \rangle \), such that \( M < M_R \), and show this leads to a contradiction.

It suffices to consider two cases: (i) \( F_R \subseteq F \) and \( T_R = T \); (ii) \( T \subseteq T_R \) and \( F_R \subseteq F \).

Case (i). \( F_R \subseteq F \) and \( T_R = T \). That \( F_R \not\subseteq F \) implies \( \Pi_{M_R} \not\subseteq \Pi_M \). Consider any clause \( r \in \Pi_M \setminus \Pi_{M_R} \). Let \( r' \) be the counterpart of \( r \) in \( \Pi \) (i.e., the clause before the transformation). Since \( M \) is a model of \( \Pi \), that \( r' \) is satisfied by \( M \) implies \( r \) is satisfied by \( M \). It follows that \( r \) is also satisfied by \( T \) (i.e., changing all the undefined to false does not change the satisfiability of a clause). Thus, \( T \) is a model of \( \Pi_M \).
Since \( T_R = T \) and \( T_R \) is a minimal model of \( \Pi_{M_R} \), \( T \) is a minimal model of \( \Pi_{M_R} \). Then, from the fact that \( \Pi_{M_R} \subseteq \Pi_M \) and \( T \) is a model of \( \Pi_M \), we have that \( T \) is also a minimal model of \( \Pi_M \). Thus, \( M \) is regular model of \( \Pi \). But \( M \) is less undefined than \( M_R \). This contradicts the assumption that \( M_R \) is a regular model.

Case (ii). \( T \subseteq T_R \) and \( F_R \subseteq F \). Clearly, since \( T_R \) is a minimal model of \( \Pi_{M_R} \), \( T \) cannot be a model of \( \Pi_M \). Then there exists a clause \( r \) in \( \Pi_M \) which is not satisfied by \( T \). It can be easily verified that the counterpart of \( r \) in \( \Pi_M \) is not satisfied by \( M \), which contradicts the assumption that \( M \) is a model of \( \Pi \).

A question arises as to under what conditions a DB has only two-valued regular models. This question is answered in the next subsection.

5.1. Program Properties under the Regular Model Semantics

In this subsection we study program properties under the regular model semantics and show a major claim mentioned earlier that the undefined is needed only for programs in which an atom negatively depends upon itself. Although the results given here are also valid for deductive databases, we restrict our technical development to logic programs.

The following definition defines a graph for a given program according to its syntactic structure. It has been called a dependency graph in the literature (see, for example, [1]).

**Definition 5.2.** Let \( \Pi \) be a program. Define a directed graph \( G_{\Pi} \) of \( \Pi \) over the set of the atoms in \( H_L \) as follows: for each clause

\[
Q \leftarrow A_1, ..., A_n, \lnot B_1, ..., \lnot B_m
\]

in \( \Pi \), place a positive arc from \( A_i \) to \( Q \), for \( 1 \leq i \leq n \); and place a negative arc from \( B_i \) to \( Q \), for \( 1 \leq i \leq m \). A path from one atom to another thus consists of a number of positive arcs and a number of negative arcs. A path from an atom to itself is called cyclic if it does not contain any identical subpath.

Note that a path obtained by composing sub-cyclic path(s) cannot be cyclic by our definition. This is to avoid ambiguity when referring to "a cyclic path from an atom to itself," because, by the above definition, no cyclic path from an atom may have the atom appear in between.
Example 5.3. The graph $G_\Pi$ where $\Pi$ consists of

\[
\begin{align*}
  b &\leftarrow \lnot a \\
  c &\leftarrow \lnot b \\
  d &\leftarrow \lnot c \\
  a &\leftarrow d
\end{align*}
\]

has a cyclic path with four arcs, three of which are negative. Because of the no repetition requirement, the cyclic path is the only one from $a$ to itself on the graph.

If a program is stratified or locally stratified [1, 9, 20], there will be no cyclic path with negative arcs in its graph. Otherwise, there will be only two types of cyclic paths, depending on whether the number of their negative arcs is even or odd. Surprisingly, this number plays an interesting role in the regular model semantics.

Definition 5.3. $\Pi$ is said to be self-contradictory if there exists in its graph a cyclic path on which the number of negative arcs is odd. Otherwise it is said to be self-contradiction free.

The intuition behind self-contradiction is that if there is a cyclic path from an atom $Q$ with an odd number of negative arcs, then it is possible that $Q$ being assigned $false$ (resp. $true$) could trigger the demand of assigning the value $true$ (resp. $false$) to $Q$, resulting in a contradiction. In Example 5.3 above, for instance, if we assume $a$ to be false, then $b$ must be true and $c$ must be false ($c$ being true cannot be justified although it does yield a model); this requires $a$ to be true.

A more detailed examination of Example 5.2 shows that the program is self-contradiction-free. As we will show, it is this property that guarantees that the regular models of the program are two-valued.

Before we show the main results of this subsection, we need one more definition regarding how ground atoms in a program can be arranged in a hierarchical fashion.

Definition 5.4. Let $\Pi$ be program. A stratification of $\Pi$ is a partial order $\leq$ over subsets of $H_L$, defined as follows:

(i) every (ground) atom belongs to one and only one stratum; and

(ii) two atoms $A$ and $B$ are in the same stratum if they are on a common cyclic path, or there exists an atom $C$ such that $A$ and $C$ are in the same stratum and the same holds true for $B$ and $C$; and these are the only atoms than can be in the same stratum.

A stratification is said to be well founded iff for every stratum \([B]\), there exists \([A]\) such that \([A] \leq [B]\) and for any stratum \([C]\), if \([C] \leq [A]\) then there are only positive arcs from atoms in \([C]\) to atoms in \([A]\). Such a stratum \([A]\) is called a base in the partial order.

The notion of stratification is well defined since it can be easily verified that given a program there always exists a stratification. A stratification may not be well founded, for example, the ground program instantiated from \(\Pi = \{p(0), p(x) \leftarrow \neg p(s(x))\}\) does not have a well-founded stratification.

The following proposition gives the main result of this subsection, which describes necessary conditions for a program to possess a nontrivial three-valued regular model and multiple regular models, respectively.

**Theorem 5.3.** Let \(\Pi\) be a program with a well-founded stratification.

(i) \(\Pi\) has a nontrivial three-valued regular model only if there is a cyclic path in the graph of \(\Pi\) with an odd number of negative arcs. The reverse, however, is not true.

(ii) \(\Pi\) possesses more than one regular model only if there exists, in its graph \(G_\Pi\), a cyclic path with an even number of negative arcs. The reverse, however, is not true.

*Proof.* See the Appendix. \(\blacksquare\)

Equivalently, we can obtain the following sufficient conditions: for programs with a well-founded stratification, if there is no cyclic path in \(\Pi\)'s graph with an odd number of negative arcs, then all the regular models of \(\Pi\) are two-valued; and if there is no cyclic path in \(\Pi\)'s graph with an even number of negative arcs, \(\Pi\) has a unique regular model.

While the condition on the existence of a well-founded stratification is absolutely needed for (ii), there is no evidence against eliminating this condition from (i). In our current proof, we have to reply on this requirement in order to have a basis to convert any three-valued justifiable model, stratum by stratum, to a two-valued justifiable model. We conjecture that this requirement can be removed from (i).

The preceding syntactic characterization of self-contradiction is not very accurate because the notion under consideration is actually a semantical one. We will further explore dynamic characterizations of a program in Subsection 5.4.

We should mention that the condition for self-contradiction-free is undecidable in the general case. There exists, however, at least one straightforward testable condition stronger than it. That is, define the graph of a program \(\Pi\) over its predicate symbols instead of ground atoms. We therefore have a finite graph with a finite number of nodes and arcs. Then, nonexistence of a cyclic path with an odd number of negative arcs is a sufficient condition for being self-contradiction-free. This strengthening is also applicable to the condition for the existence of multiple regular models.
5.2. Relationship with the Well-Founded Semantics

The well-founded model assigns the truth value false only to those atoms that are called *unfounded* [10] while the regular models allow negative literals to be *assumed* if such assumptions do not lead to any contradiction. As a matter of fact, atoms that are true (resp. false) in the well-founded model of a program $\Pi$ must be true (resp. false) in every regular model of $\Pi$.

**Theorem 5.4.** Every regular model of a program $\Pi$ is an extension of the well-founded model of $\Pi$.

**Proof.** See the Appendix. 

Indeed, the regular model semantics is a refinement of the well-founded semantics with less atoms being treated as undefined. The following corollary directly follows from Theorem 5.4.

**Corollary 5.5.** Let $\Sigma$ be the set of all regular models of a program $\Pi$, and let $N = \langle T_N; F_N; U_N \rangle$ be the well-founded model of $\Pi$. Let $M = \langle T_M; F_M; U_M \rangle$ be the intersection of all regular models in $\Sigma$; i.e., $M = \cap_{M_i \in \Sigma} M_i$. Then, $T_N \subseteq T_M$ and $F_N \subseteq F_M$.

A proof-theoretic implication of these results is that, when dealing with queries composed of a conjunction of literals, any sound proof procedure with respect to the two-valued logic (see, for example, [26]) for the well-founded semantics is also sound for the regular model semantics. The completeness, however, is not preserved in general, because, as shown by the following two examples, the intersection of all regular models of a program need not be equivalent to its well-founded model.

**Example 5.4.** Let $\Pi$ consist of

\[
\begin{align*}
b & \leftarrow \neg a \\
a & \leftarrow \neg b \\
\end{align*}
\]

The only regular model is $M_1 = \langle \{a\}; \{b\} \rangle$, while the well-founded model is $M_2 = \langle \emptyset; \emptyset \rangle$. The reason for the discrepancy is that the "second half" of the disjunctive information by the first two clauses, i.e., $b$ is true and $a$ is false, has been prevented from forming a model by $a$ negatively depending upon itself. Note, however, that in any case the intersection of all regular models of a program $\Pi$ contains more information than the well-founded model of $\Pi$.

**Example 5.5** (Van Gelder 1988 [10]). Let $\Pi$ be

\[
\begin{align*}
b & \leftarrow \neg a \\
\end{align*}
\]

\[
\begin{align*}
a & \leftarrow \neg b \\
p & \leftarrow a \\
p & \leftarrow b.
\end{align*}
\]
Its well-founded model has all atoms \( a, b, \) and \( p \) undefined while its two regular models are: \( M_1 = \langle \{a, p\}; \{b\} \rangle \) and \( M_2 = \langle \{b, p\}; \{a\} \rangle \), whose intersection is \( \langle \{p\}; \emptyset \rangle \) which is not even a justifiable model. This discrepancy stems from the inability of the well-founded approach to perform reasoning by cases.

5.3. Relationship with Stable Models

There is a one-to-one correspondence between two-valued regular models and stable models of a program. A nontrivial three-valued regular model corresponds to a "disappeared stable model" because of the existence of the "difficult-to-be-assigned" atoms.

**Theorem 5.6.** Any two-valued regular model of a program \( \Pi \) is a stable model of \( \Pi \), and vice versa. Consequently, \( \Pi \) does not have a stable model iff its regular models are all nontrivially three-valued.

**Proof.** It follows from the fact that for any two-valued model \( M \), \( M \) is stable model iff it is justifiable. 

Together with Theorem 5.3, we obtain a sufficient condition for the existence of a stable model. The condition can be considered syntactic as it is completely based on the structure of the given program.

**Theorem 5.7.** Let \( \Pi \) be a program with a well-founded stratification. \( \Pi \) has a stable model if there is no cyclic path in the graph of \( \Pi \) with an odd number of negative arcs.

As mentioned at the outset of this paper, the reason for the non-existence of a stable model is because of "difficult-to-be-assigned" atoms. The effect of these atoms is global in the stable model semantics but local in the regular model semantics. This is illustrated in the following example.

**Example 5.6 (Van Gelder, 1988 [10]).** Consider

\[
\begin{align*}
b & \leftarrow \neg a \\
a & \leftarrow \neg b \\
p & \leftarrow \neg p \\
p & \leftarrow \neg a.
\end{align*}
\]

This program has two regular models: \( M_1 = \langle \{b, p\}; \{a\} \rangle \) and \( M_2 = \langle \{a\}; \{b\} \rangle \). \( M_1 \) is the unique stable model of the program while \( M_2 \) describes a consistent explanation of the program by localizing the effect of the difficulty of assigning \textit{true} or \textit{false} to \( p \).

We now give the relationship among the well-founded, regular, and stable model semantics.
THEOREM 5.8. Let $\Sigma$ be the set of all regular models of a program $\Pi$ and $\Gamma$ the set of all its stable models. Assume further that $\Gamma$ is nonempty. Let $N = \langle T_N; F_N; U_N \rangle$ be the well-founded model of $\Pi$, $M = \langle T_M; F_M; U_M \rangle$ be the intersection of all regular models in $\Gamma$, where $T_S = \bigcap_{M_i \in \Gamma} M_i$ and $F_S = H_L - \bigcup_{M_i \in \Gamma} M_i$. Then, $T_N \subseteq T_M \subseteq T_S$ and $F_N \subseteq F_M \subseteq F_S$.

Proof. It follows from Theorems 5.5 and 5.6.

5.4. When Is Undefined Really Needed?

We have seen that based on the justifiability principle, undefined is needed in order to avoid nonexistence of an intended model for some programs. The following theorem says that any nontrivial three-valued regular model must involve an undefined atom that occurs in a cyclic path of an odd number of negative arcs.

THEOREM 5.9. Let $\Pi$ be a program with a well-founded stratification and $G_\Pi$ its graph. Suppose that $M_R$ is a nontrivial three-valued regular model of $\Pi$. Then, there is an atom which is undefined in the model and which is on a cyclic path with an odd number of negative arcs.

Proof. Assume that this is not true; i.e., all the atoms involved in some cyclic path are assigned either $t$ or $f$. Then, there is always a way to assign $t$ or $f$ to the other undefined such that the resulting interpretation is a justifiable model (see the proof of part (i) of Theorem 5.3 in the Appendix for the claim). Since the resulting model is also two-valued, it must be less undefined than $M_R$. This contradicts the assumption that $M_R$ is regular. Therefore, the statement in the theorem must hold.

COROLLARY 5.10. Let $\Pi$ be a program with a well-founded stratification. $\Pi$ possesses a stable model iff the atoms involved in any cyclic path with an odd number of negative arcs are assigned $t$ or $f$ in every regular model of $\Pi$.

Proof. It follows from Theorems 5.6 and 5.9.

The above corollary explains why the syntactic condition given in Theorem 5.7 is not accurate.

EXAMPLE 5.7 (Lifschitz, 1988 [16]). Let $\Pi$ consist of

\[ p(1, 2) \]
\[ p(2, 1) \]
\[ q(x) \leftarrow p(x, y), \neg q(y). \]

This program is self-contradictory if instantiated over the entire Herbrand universe. However, it is impossible to get into the situation where we have $q(1) \leftarrow \neg q(1)$ or $q(2) \leftarrow \neg q(2)$ (which would be the case if $p(1, 1)$ or $p(2, 2)$ were
true), resulting in either (or both) of them becoming undefined. The only two regular models are two-valued and therefore stable models:

\[ M_1 = \{ p(1, 2), p(2, 1), q(1) \} \]
\[ M_2 = \{ p(1, 2), p(2, 1), q(2) \}. \]

Lifschitz [16] pointed out that one perhaps should not attempt to define a semantics for such a logic program based on selecting a single model.

The implication of these results (plus the conjecture we made earlier in Subsection 5.1 about being able to remove the requirement of the existence of a well-founded stratification) is that the situation where an atom negatively depends upon itself dynamically, possibly in a cascaded fashion, in the only cause for the need of the undefined. Taking, for example, the two-person game program from [10]):

\[ \text{win}(y) \leftarrow \neg \text{win}(x), \text{move}(x, y). \]

The situation where an atom negatively depends on itself may arise only when no move is allowed, i.e., \( \text{move}(x, x) \) is true for some \( x \) in some regular model. In that case, however, the clause can no longer justify a winner according to the principle of justifiability. It is debatable whether the intuitive reading of the clause verifies \( \text{winner}(x) \), or yields the conclusion that a draw has been reached. It appears that programs lying outside this condition are often unclear in their intended semantics, if not completely ambiguous.

**APPENDIX**

We first introduce the concept of *quasi-clause* and then show that every program has an equivalent quasi-clause program. In addition to having subbstantially simplified the proofs needed in this paper, the material presented here is of interest in its own right.

6.1. Quasi-Clause Programs

**Definition 6.1.** A clause is said to be a *quasi-clause* if no positive literals appear in the body. It is otherwise called a non-quasi-clause. A program is said to be a *quasi-clause program* if all the clauses therein are quasi-clauses.

We have seen that the well-founded, stable, and regular model semantics all require the justifiability principle. Justifiability can be equivalently described as repeatedly reducing positive atoms in the body of a clause using all their definition clauses. A reduced clause where a positive atom either cannot be further reduced or is involved in some infinite reduction process without being able to be fully reduced should then be disregarded. The following definition describes a reduction procedure for generally non-ground logic programs.
DEFINITION 6.2. Let $\Pi$ be a (ground or nonground) logic program. Let $R$ be a selection rule such that, given a conjunction of literals, it always selects a positive literal therein; $R$ is not applicable if the conjunction contains only negative literals.

Let $C$ be a clause in $\Pi$. A quasi-tree for clause $C$ is built as follows. The nodes of the tree are clauses. The immediate descendent nodes of a node $Q \leftarrow A_1, ..., A_n, \neg B_1, ..., \neg B_m$ are obtained by resolving with a positive literal $A_i$, selected by $R$, for each of the definition clauses $D \leftarrow L_1, ..., L_k$ in $\Pi$ of $A_i$ such that $D$ and $A_i$ are unifiable with $\sigma$ being the most general unifier. That is, a descendent node is of the form

$$\sigma Q \leftarrow \sigma(A_1, ..., A_{i-1}, L_1, ..., L_k, A_{i+1}, ..., A_n, \neg B_1, ..., \neg B_m).$$

A branch in the tree is either infinite or ends with a leaf node. A leaf node is either a quasi-clause or a non-quasi-clause. (A non-quasi-clause leaf node therefore contains at least one positive literal that cannot be further resolved upon.) Let $\Gamma$ be the set of all quasi-clause leaf nodes and

$$\Pi' = \Pi - \{C\} \cup \Gamma.$$  

$\Pi'$ is called a reduced program of $\Pi$. Such reduction is repeatedly performed for each of the remaining non-quasi-classes in $\Pi'$. When all clauses have been reduced, the resulting program contains only quasi-clauses and is denoted as $QUA(\Pi)$.

EXAMPLE 6.1. Consider the program in Example 5.3 again:

$$b \leftarrow \neg a$$
$$c \leftarrow \neg b$$
$$d \leftarrow \neg c$$
$$a \leftarrow d.$$  

The positive atom $d$ in the last clause can be reduced by using the third rule, resulting in the transformed program:

$$b \leftarrow \neg a$$
$$c \leftarrow \neg b$$
$$d \leftarrow \neg c$$
$$a \leftarrow \neg c.$$  

Note that a quasi-tree actually describes the first part of the proof procedures in [24, 25] (also see [27]) for the well-founded semantics; i.e., the positive literals in a goal are always resolved before the negative literals in the goal, and if some positive literal therein cannot be resolved to the empty clause, the goal fails and its truth value is false.
The idea of quasi-clauses should be traced back to the method of fixpoint completion, introduced by Dung and Kanchanasut [5], who provided an elegant bottom-up version of the above described transformation (plus an application of predicate completion) and showed precisely the difference between the completion semantics and minimal model-based semantics such as the stable model semantics.

6.2. Relation between Well-Founded and Fitting-Kunen Semantics

We show two results in this subsection: (i) for quasi clause programs, the well-founded model coincides with the least fixpoint of the Fitting-Kunen operator [8,15], and (ii) a program and its quasi-clause program have the same well-founded model.

The Fitting-Kunen operator computes both positive and negative atoms directly from previously known positive and negative facts. Their operator is monotonic, and thus the least fixpoint exists.

The fact that Przymusinski's operator $I$ (see Subsection 2.2.1) is the same mapping as that of Fitting and Kunen's for quasi-clause programs is shown as follows. First, because no positive literals appear in the body of any clause, the mappings $T_I$ and $\Phi_I$ in Przymusinski's framework reduce to

$$T(\langle T; F \rangle) = \{ A \mid \text{there exists a clause } A \leftarrow \neg C_1 \land \cdots \land \neg C_n \text{ such that } C_i \text{ are in } F \};$$

$$\Phi(\langle T; F \rangle) = \{ A \mid \text{for every clause } A \leftarrow \neg C_1 \land \cdots \land \neg C_n, \text{ there exists a } C_j \in T \}.$$

These two operators can be easily merged into a single operator:

$$I_H(\langle T; F \rangle) = \langle T'; F' \rangle,$$

where

$$T' = T \cup T(\langle T; F \rangle) \quad \text{and} \quad F' = F \cup \Phi(\langle T; F \rangle).$$

Now, it can be seen that the two iterative processes in Przymusinski's formalism, first by repeatedly applying $T_I$ and $\Phi_I$ and then by $I$, is equivalent to the following iterative process:

$$I_H^0 = \langle \emptyset; \emptyset \rangle \quad \text{and} \quad I_H^{n+1} = I_H(I_H^n).$$

The operator $I_H$ defined above is exactly the same as the Fitting–Kunen operator. We therefore have shown:

**Proposition 6.1.** Let $\Pi$ be a quasi-clause program. Then, the Fitting–Kunen operator $I_H$ coincides with Przymusinski's operator $I$. Therefore, the least fixpoint computed by the former is the well-founded model of $\Pi$. 
It should be clear from the definitions that the quasi-clause transformation described in Subsection 6.1 has no effect on the atoms that are true or false in the well-founded model of program $\Pi$ (see Subsection 2.2.1); i.e., an atom $Q$ is true (resp. false) in the well-founded model of $\Pi$ iff it is true (resp. false) in the well-founded model of $QUA(\Pi)$. We thus have the following proposition:

**Proposition 6.2.** Let $\Pi$ be a program and $QUA(\Pi)$ be its quasi-clause program. Then, the well-founded model of $\Pi$ coincides with the well-founded model of $QUA(\Pi)$.

### 6.3. Relation between $\Pi$ and $QUA(\Pi)$

We now explain in what sense a transformed quasi-clause program is equivalent to its original one. One small technical problem is that a model of $QUA(\Pi)$ may not necessarily be a model of $\Pi$, because of the disappeared atoms.

**Example 6.2.** Let $\Pi = \{\neg a \rightarrow b; c \leftarrow d, c\}$. Then, $QUA(\Pi) = \{\neg a \rightarrow b\}$ with both $c$ and $d$ disappearing. Although interpretation $\langle \{b, c\}; \{a, d\} \rangle$ is a model of $QUA(\Pi)$, it is not a model of $\Pi$ because the assignment of $c$ and $d$ does not satisfy $\Pi$ though it does satisfy $QUA(\Pi)$.

**Definition 6.3.** A model of $QUA(\Pi)$ is a three-valued interpretation $M = \langle T; F \rangle$ of $\Pi$ such that every clause in $QUA(\Pi)$ is satisfied in $M$, and for the subset $M'$ of $M$ which interprets those predicates in $\Pi$ but not in $QUA(\Pi)$, there is a model $M''$ of $\Pi$ such that $M' \subseteq M''$.

The definition says that the predicates that are in $\Pi$ but not in $QUA(\Pi)$ should be interpreted, in a model of $QUA(\Pi)$, to satisfy $\Pi$.

**Proposition 6.3.** Let $\Pi$ be a program and $QUA(\Pi)$ its quasi-clause program. Then,

(i) $QUA(\Pi)$ and $\Pi$ have the same set of models; and therefore the same set of minimal models; and

(ii) $QUA(\Pi)$ and $\Pi$ have the same set of justifiable models; in particular, they have the same set of stable models, and the same set of regular models.

**Proof:** It is clear that any model of $\Pi$ is a model of $QUA(\Pi)$. The reverse is guaranteed by Definition 6.3. This proves (i). It is also clear that the transformation defined in Definition 6.2 does not change justifiability; in particular, the deleted non-quasi-clauses in a quasi-tree cannot be used to justify any atom. This proves (ii).

Now, as far as cyclic paths are concerned, it can be seen that any cyclic path in the graph of $QUA(\Pi)$ must be a projection of a cyclic path in the graph of $\Pi$ with
all positive arcs removed. For example, consider Example 6.1 again. The cyclic path from \( a \) to itself in \( G_{QUA(II)} \), where \( QUA(II) \) is

\[
\begin{align*}
b & \leftarrow \neg a \\
c & \leftarrow \neg b \\
d & \leftarrow \neg c \\
a & \leftarrow \neg c,
\end{align*}
\]
is a projection from that in \( G_H \) with the positive arc removed. In general, however, a cyclic path in \( G_H \) may disappear in \( G_{QUA(II)} \) due to the removal of positive arcs. The following definition makes this precise.

**Definition 6.4.** Let \( A \leftarrow\neg B \) denote a negative arc from node \( A \) to node \( B \) in a graph, and let \( A \leftarrow C \) denote zero or more positive arcs from \( A \) to \( C \).

Let \( L_p \) and \( L'_p \) be two paths in two graphs, respectively. \( L'_p \) is said to be a projection of \( L_p \) if \( L'_p \) can be obtained by replacing one or more subpath(s) of the form \( A \overset{\rightarrow}{\rightarrow} B \overset{\rightarrow}{\rightarrow} C \) or \( A \overset{\rightarrow}{\rightarrow} B \rightarrow C \) with \( A \overset{\rightarrow}{\rightarrow} C \). The projection relation is reflexive and transitive. That is, any cyclic path \( L_p \) is its own projection; and if \( L_p \) is a projection of \( L_p' \) and \( L_p' \) a projection of \( L_p'' \), then \( L_p \) is a projection of \( L_p'' \).

The following proposition directly follows from the preceding definition and the definition of the quasi-clause transformation.

**Proposition 6.4.** Let \( II \) be a program and \( QUA(II) \) its quasi-clause program. Then, any cyclic path in the graph of \( QUA(II) \) consists of negative arcs only and must be a projection of a cyclic path in the graph of \( II \).

Finally, it can be easily verified that a program \( II \) has a well-founded stratification iff \( QUA(II) \) has a well-founded stratification. Because of this and Propositions 6.2, 6.3, and 6.4, it is sufficient to carry out our proofs only for quasi-clause programs. From now on, when we say a program, we mean a countable set of ground quasi-clauses which may well be infinite.

**6.4. Proof of Theorem 5.3**

To capture the effect of a cyclic path in the graph of a program, we can manipulate the program clauses as follows: for any two consecutive arcs from \( C_1 \) to \( A \) via \( B_1 \) in the graph and their corresponding clauses,

\[
\begin{align*}
A & \leftarrow \neg B_1, \ldots, \neg B_{n_1} \\
B_1 & \leftarrow \neg C_1, \ldots, \neg C_{n_2},
\end{align*}
\]
replace \( B_1 \) in the first clause by the body of the second. The body of the resulting clause is then equivalent to a disjunctive normal form,

\[
A \leftarrow (\Gamma \land C_1) \lor \cdots \lor (\Gamma \land C_{n_2}),
\]
where $F = \neg B_2 \land \cdots \land \neg B_n$. We will call this reduction process and its repeated applications \textit{backward reduction} on $A$ and $B_1$ a \textit{connecting atom} of $A$. Backward reduction may be simultaneously applied to each applicable connecting atom using a definition clause of the atom. We call this \textit{parallel backward reduction}. Note that each backward reduction sequence corresponds to a path in the graph. Thus, for every cyclic path there is a backward reduction sequence.

\textbf{Lemma 6.5.} Let $\Pi$ be a quasi-clause program. Consider any path from an atom $B$ to an atom $A$ of length $n$ that has no $B$ appear in between or in the body of any definition clause of $A$. If $n$ is an even number, then backward reduction on $A$ along the path yields a formula

$$A \leftarrow M,$$

where $M$ is a disjunctive normal form in which $B$ appears positively; if $n$ is odd, then backward reduction on $A$ along the path yields a formula

$$A \leftarrow N,$$

where $N$ is a disjunctive normal form in which $B$ appears negatively.

\textit{Proof.} By an easy induction on the number of arcs on a path.

This lemma explains why an atom might be "difficult-to-be-assigned" if it is involved in a cyclic path with an odd number of negative arcs. In that case, we will get a formula of the form

$$A \leftarrow (\Theta_1 \land \neg A) \lor \Theta_2,$$

where $\Theta_1$ and $\Theta_2$ are some formulas. If $\Theta_2$ is false and $\Theta_1$ is true, we then virtually have $A \leftarrow \neg A$. If $A$ cannot be justified by some other clause, than $A$ can neither be true nor false by the justifiability principle, and thus it has to be undefined.

The next lemma says that any atom involved in some cyclic path in its graph can be represented equivalently in a certain form through the use of parallel backward reduction. Let us first see an example.

\textbf{Example 6.3.} Consider the following program

$$b \leftarrow \neg e, \neg a$$
$$a \leftarrow \neg d, \neg b$$
$$d \leftarrow \neg a, \neg e$$
$$e \leftarrow \neg b.$$

This program has a well-founded stratification: the stratum at the bottom is $\{e\}$ and the next higher stratum contains all the rest of the atoms. Consider parallel
backward reduction on \( b \) with respect to the two cyclic paths issuing from \( b: \{ b, a, b \} \) and \( \{ b, e, d, a, b \} \). The clause \( b \leftarrow \neg c, \neg a \) is first reduced to \( b \leftarrow (\neg c \land d) \lor (\neg c \land b) \). The reduction with respect to the first cyclic part is done and is represented by the second conjunct. This clause is further reduced, with respect to the second cyclic path, to \( b \leftarrow (\neg c \land \neg a \land \neg e) \lor (\neg c \land b) \) and then to \( b \leftarrow (\neg c \land \neg a \land b) \lor (\neg c \land b) \). Note that \( b \) appears in each of the conjuncts. Note also that there are alternatively cyclic paths issuing from \( b \) which can make \( b \) appear in both conjuncts.

In general, a connecting atom may have more than one definition clause and thus parallel backward reduction should be applied using all alternative definition clauses. For the purpose of justifiability, we can combine all the definition clauses of an atom \( A \leftarrow \Theta_1, A \leftarrow \Theta_2, \ldots, \) and \( A \leftarrow \Theta_n \), into one with a disjunctive body \( A \leftarrow \Theta_1 \lor \cdots \lor \Theta_n \). We assume in the following lemma that when parallel backward reduction is applied such a conversion is implied implicitly.

**Lemma 6.6.** Let \( \Pi \) be a quasi-clause program. Let \([A]\) be a stratum in a stratification of \( \Pi \) which contains cyclic paths with an even number of negative arcs only. Then, there exists a parallel reduction sequence with respect to some cyclic paths issuing from \( A \), which yields a formula \( A \leftarrow M \) such that \( M \) is a disjunctive normal form where \( A \) appears only positively and appears once in each of the conjuncts in which an atom from \([A]\) appears.

**Proof.** By Lemma 6.5, if \( A \) does not appear negatively in a definition clause of \( A \), the backward reduction with respect to a single cyclic path, starting from \( A \), with an even number of negative arcs results in a formula in the disjunctive normal form, where \( A \) appears only positively. So, \( A \) cannot possibly appear negatively in the formula. Let \( M = \Gamma_1 \lor \cdots \lor \Gamma_n \). If some conjunct \( \Gamma_k \) in \( M \) contains an atom \( Q \in [A] \) and has no \( A \) appearing in it, then the parallel backward reduction is not completed because, by the definition of a stratum, there is a path from \( A \) to \( Q \) and a path from \( Q \) to \( A \).

We are now ready to prove Theorem 5.3.

**Theorem 5.3.** Let \( \Pi \) be a countable set of ground quasi-clauses with a well-founded stratification.

(i) \( \Pi \) has a nontrivial three-valued regular model only if there is a cyclic path in the graph of \( \Pi \) with an odd number of negative arcs. The reverse is not true.

(ii) \( \Pi \) possesses more than one regular model only if there exists, in its graph \( G_\Pi \), a cyclic path with an even number of negative arcs. The reverse is not true.

**Proof of (i).** We show that if there is no cyclic path in the graph of \( \Pi \) with an odd number of negative arcs, then every regular model of \( \Pi \) is two-valued. We
show this by constructing a two-valued model from an arbitrary three-valued justifiable model such that the two-valued is less undefined than the three-valued.

Let $M_s = \langle T_s; F_s; U_s \rangle$ be a justifiable model of $\Pi$. Assume there is no cyclic path in $G_\Pi$ with an odd number of negative arcs. We show that there exists an assignment of $\text{t}$ or $\text{f}$ to any $A$ in $U_s$ such that the resulting interpretation is a justifiable model of $\Pi$.

Given a well-founded stratification of $\Pi$, we proceed the assignment of the undefined as follows. If $[A] \leq [B]$, then the assignment of the undefined in $[A]$ is done before those in $[B]$. Since the stratification is well founded, this is always possible. Thus, when the undefined in $[B]$ are being assigned, all the atoms in lower strata have already been assigned.

For each stratum that contains the undefined, the assignment of the atoms not involved in any cyclic path is straightforward because of the lower strata have already been assigned. (These atoms are undefined simply because some of their definition clauses contain at least one undefined atom from a lower stratum.)

Otherwise, for each undefined atom $A$ in the current stratum that is involved in some cyclic path, apply parallel backward reduction with respect to all the cyclic paths in which it is involved. By Lemma 6.6, we will get a formula $A \leftarrow M$ such that $M$ is a disjunctive normal form where $A$ appears only positively and appears at least in each of the conjuncts in which an atom from $[A]$ appears. Clearly, a conjunct that does not contain $A$ consists of atoms all of which are from some lower strata and thus are already assigned. Now, let $A_1, \ldots, A_m$ be the unassigned atoms in the current stratum. We thus have the formulas:

$$A_1 \leftarrow M_1, \quad A_2 \leftarrow M_2, \quad \ldots, \quad A_m \leftarrow M_m.$$ 

The assignment of these $A_i$'s can proceed sequentially in an arbitrary order in the following way: if a conjunct in the body of $A_i \leftarrow M_i$ evaluates to $\text{true}$ in the current interpretation (under construction), then assign $\text{t}$ to $A_i$; otherwise assign $\text{f}$ to $A_i$.

By a straightforward induction over the strata in the stratification, we can show that, for every $i$, (i) every undefined will be assigned $\text{t}$ or $\text{f}$, (ii) all the program clauses involving an undefined atom $A_i$ will be satisfied after the assignment, and (iii) if an undefined atom $A_i$ is assigned $\text{t}$ this way; then $A$ being true is justifiable simply because backward reduction preserves justifiability. Thus, the resulting two-valued interpretation is a justifiable model of $\Pi$.

To show that the reverse is not true, consider program $P = \{a \leftarrow b \leftarrow \neg a, \neg b\}$. There is, in its graph, a cyclic path with one negative arc. However, it has a unique regular model with an empty set of the undefined: $\langle \{a\}; \{b\} \rangle$. This completes the proof of (i).

Proof of (ii). Let $M_{R_1} = \langle T_{R_1}; F_{R_1} \rangle$ and $M_{R_2} = \langle T_{R_2}; F_{R_2} \rangle$ be two distinct regular models of $\Pi$. Then, there exists an atom $Q$ such that either $Q \in T_{R_1}$ and $Q \notin T_{R_2}$, or $Q \in F_{R_1}$ and $Q \notin F_{R_2}$. We show the first case and omit the second as it is similar.

Since $Q \in T_{R_1}$, there is a clause $Q \leftarrow \neg B_1, \ldots, \neg B_n, n \geq 0$, such that $B_i \in F_{R_1}$, for each $i \leq n$. Hence, for each $B_i$ and each definition clause $B_i \leftarrow \neg C_1, \ldots, \neg C_k$, there
exists $C_j \in T_{R_1}$, $1 \leq j \leq k$. This argument continues for each of these $C_j$ and so on. Eventually we will face two situations: no involvement of any cyclic path and otherwise.

Case 1. No cyclic path reaching $Q$. In this case, because of the existence of a well-founded stratification, eventually a true atom $D$ is justified by a clause with an empty body, or by a set of literals \{-$D_1$, ..., -$D_n$\} such that each $D_i$ therein is in a base stratum and therefore should be false. Thus, $Q$ must be true in every regular model; this contradicts the assumption $Q \notin T_{R_2}$. Hence, this case is not possible.

Case 2. Existence of a cyclic path reaching $Q$. Such a cyclic path is of the form:

$$A_0 \leftarrow \ldots, \neg A_1, \ldots$$

$$A_1 \leftarrow \ldots, \neg A_2, \ldots$$

\ldots

$$A_q \leftarrow \ldots, \neg A_0, \ldots,$$

such that, for every $i \leq q$, if $A_i \in T_{R_1}$ then $A_{i+1} \in F_{R_1}$, modulo $q$, and there is a path (of length 0 or more) from $A_0$ to $Q$. Clearly, this would not have been possible if $q$ were an odd number, because in that case, $A_i \in T_{R_1}$ would require $A_{i-1} \in F_{R_1}$. Therefore, $q$ must be an even number. See Example 5.2 for the falsity of the reverse. 

6.5. Proof of Theorem 5.4

**Theorem 5.4.** Every regular model of a program $\Pi$ is an extension of the well-founded model of $\Pi$.

To prove this claim, we need the following two lemmas.

**Lemma 6.7.** Let $M_w = \langle T_w; F_w \rangle$ be the well-founded model of a program $\Pi$ and $M_R = \langle T_R; F_R \rangle$ a regular model of $\Pi$. Then, $T_w \cap F_R = T_R \cap F_w = \emptyset$.

**Proof.** We prove that $T_w \cap F_R = \emptyset$ by assuming that there is $Q \in T_w \cap F_R$ and showing that this will lead to a contradiction. The proof of $T_R \cap F_w = \emptyset$ is similar and thus omitted.

Since $Q \in T_w$ and $M_w$ is justifiable, there exists a clause $Q \leftarrow \neg B_1, \ldots, \neg B_n$ in $\Pi$ such that $B_i \in F_w$ for all $i \leq n$. Again, by the definition of fixpoint computation of well-founded model, for each clause $B_i \leftarrow \neg C_1, \ldots, \neg C_k$, there exists $C_j \in T_w$, and so on.

Since there are only finite steps in justifying any atom, eventually, by the coincidence of the well-founded model and the least fixpoint of the Fitting–Kunen operator (Proposition 6.1), such a $B_i \in F_w$ is one that has no definition clause and such a $C_j \in T_w$ is just a clause.
Now consider the same justification sequence above for $M_R$. Since $Q \in F_R$, at least one of the $B_i$'s above must be in $T_R$, in order to satisfy the clause. Since $M_R$ is justifiable, there is at least one of the clauses $B_i \leftarrow \neg C_1, \ldots, \neg C_k$ such that $C_j \in F_R$ for all $j \leq k$. However, the justification sequence for $Q \in T_W$ has led to the situation where such a $B_i \in F_R$ is one that has no definition clause and such a $C_j$ is just a clause. Thus $M_R$ cannot be a justifiable model.

**Lemma 6.8.** Let $M_w = \langle T_w; F_w \rangle$ be the well-founded model of a program $\Pi$ and $M_R = \langle T_R; F_R \rangle$ a regular model of $\Pi$. Then, $M_w \cup M_R$ is a model of $\Pi$.

**Proof.** We show that if a clause in $\Pi$ is satisfied by both $M_w$ and $M_R$, respectively, then it is also satisfied by $M_w \cup M_R$.

By Lemma 6.7, we only need to consider three ways that a clause is satisfied in $M_w$ or in $M_R$, respectively: $t \leftarrow u$, $u \leftarrow f$, and $u \leftarrow u$. That is, only the undefined in $M_w$ or in $M_R$ may change (to true or false) in their union.

The first two cases are trivially satisfied no matter how $u$ is changed. For the third case $u \leftarrow u$, the only possibilities that the clause can become unsatisfied are: $u \leftarrow t$, $f \leftarrow t$, and $u \leftarrow f$. By Lemma 6.7 again, it is easy to see that none of them is possible. Since every clause in $\Pi$ is satisfied by $M_w \cup M_R$, $M_w \cup M_R$ is therefore a model of $\Pi$.

**Proof of Theorem 5.4.** Let $M_w = \langle T_w; F_w \rangle$ be the well-founded model of a program $\Pi$ and $M_R = \langle T_R; F_R \rangle$ a regular model of $\Pi$. Assume that $M_R$ is not an extension of $M_w$. First, by Lemma 6.8, $M_w \cup M_R$ is a model of $\Pi$. Using Lemma 6.7, it is straightforward to show that any atom in $T_R$ or in $T_w$ is justifiable in $M_R \cup M_w$. Thus, $M_w \cup M_R$ is a justifiable model of $\Pi$. However, $M_w \cup M_R <_{\text{undef}} M_R$. This contradicts the assumption that $M_R$ is regular. Therefore, $M_R$ must be an extension of $M_w$.

**References**

24. T. PRZYMUSINSKI, Every logic program has a natural stratification and an iterated least fixed point model, in "Proceedings, the 8th ACM PODS, 1989," pp. 11–21.