A transformation system for deductive database modules with perfect model semantics*

Michael J. Maher

IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

Communicated by G. Levi
Received September 1990
Revised September 1991

Abstract


We present a transformation system for deductive database (DDB) modules. We show that it preserves several data-dependency properties of a DDB and is correct for the "perfect model" semantics of DDBs. Perfect models are not directly amenable to logical reasoning since logically equivalent DDBs may have different perfect models. We develop an approach which involves using a condition on data dependencies in DDBs (stratification compatibility) to pass from a logical equivalence to equivalence under perfect model semantics. This is readily applicable to the transformation system.

1. Introduction

The perfect model (or standard model) semantics of stratified deductive databases [18, 2] has now become widely accepted. The semantics has a natural and intuitive interpretation in terms of finishing the evaluation of a predicate before the complement of the predicate is used. Other semantics for larger classes of deductive databases (DDBs), such as the well-founded semantics [30] and the stable model semantics [13], agree with the perfect model semantics on stratified DDBs. Furthermore, the perfect model semantics is used in current DDB implementations (for

Correspondence to: M.J. Maher, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, Email: mjm@watson.ibm.com.

example, see [9]). However, the use of a semantics based upon a single model chosen by the syntactic structure of the DDB can create some problems. For example, logically equivalent DDBs may have different perfect models. This makes it difficult to straightforwardly exploit the considerable body of work on reasoning in first-order logic when optimizing or reasoning about queries using this semantics, despite the apparent close relation to logical notation.

The first result of this paper shows that, under the condition of stratification compatibility, two DDBs have the same perfect model if they are logically equivalent. More generally, it shows that for any semantics $S$ which is coarser than perfect model semantics if two stratification compatible DDBs are equivalent under $S$ then the DDBs have identical perfect models. (Here we take a semantics for a DDB to be the association of a set of models to the DDB, and we say that one semantics $S_1$ is coarser than another $S_2$ if, for every program $Q$, $S_1(Q) \subseteq S_2(Q)$.)

The second part of the paper exploits this result to show the correctness of a transformation system with respect to the perfect model semantics. Such a transformation system can be viewed as a framework for performing optimizations of DDBs. Specifically, we independently show correctness with respect to the completion semantics [10], and the stratification compatibility of DDBs which differ by only a single transformation. The previous result allows us to conclude the correctness for perfect model semantics.

We also demonstrate that the transformation system preserves several dependency properties [15], even if the DDB acted upon does not have a perfect model semantics. These dependency properties have been used previously to define classes of programs for which SLDNF resolution is complete [15, 6] and, more recently, in the study of stable models [23, 31].

The transformation system contains Unfold, Fold and Replacement transformations, among others. It is an extension of the transformation system discussed in [16, 17]. It differs fundamentally from the transformation system introduced by Tamaki and Sato [27] and extended by many others. In addition, there are two complications to the transformation system. We allow constraints to be used in the DDB, and the transformation system operates on modules rather than on entire DDBs.

The constraints are treated as in the CLP scheme [14], that is, our DDB language, transformation system and the results are all parameterized by the choice of a (generally many-sorted) data domain and the class of constraints which are allowed in DDBs. The extension of perfect model semantics to this scheme is relatively straightforward, but the formulation of the transformations is more difficult than for the usual Herbrand domain.

The parameterized treatment means that our results will hold for quite complex data domains and constraints, so that we are essentially treating programming languages as well as DDB languages. In most programming language cases the perfect model semantics is not computable, although for restrictions on the class of programs it can be (see, for example, [1]). Nevertheless, the transformations are computable
(provided certain basic operations on constraints are computable), and the results continue to hold, irrespective of computability.

A module system would seem to be necessary in a DDB with a large deductive component. We consider a class of module systems in which: modules can hide predicates, but neither function symbols nor constraints; predicates are defined in a single module; there can be no recursion between modules, only within modules. Although this class is somewhat restrictive, the restrictions reflect sensible programming practices. We show that several modules may be transformed simultaneously, and modules may be combined, without affecting the semantics of the network of modules which contains them.

The next section provides some preliminary definitions. Following that we present definitions of program dependencies and introduce stratification compatibility. We show that, for any semantics coarser than perfect model semantics, if two programs are equivalent and stratification-compatible then they have identical perfect model semantics. In Section 4 we outline the kind of modules and module composition that we treat, define equivalence of modules, and extend the previous result from programs to modules. Section 5 presents the transformations. In Section 6 we show that the transformation system preserves the perfect model semantics. We also show that many of the program-dependency properties are preserved. We conclude by briefly discussing the extensions of this work.

2. Preliminaries

We use the symbols \( \Sigma \) and \( V \) to denote, respectively, the collection of function symbols and the infinite collection of variables. Terms are constructed from \( \Sigma \) and \( V \) in the usual way. The predicate symbols are partitioned into two sets: \( P_U \), which are the predefined predicates and \( P_U \), which are the predicates to be defined by the program. We assume that \( P_U \) contains the predicate symbol \( = \). The language of all first-order formulas built from these symbols is denoted by \( L \). We use \( \equiv \) to denote syntactic identity. For any expression \( e \), \( \text{var}(e) \) denotes the set of free variables of \( e \).

We assume throughout that there is an intended domain of computation \( \mathcal{O} \). The structure \( \mathcal{O} \) defines the set \( D \) of elements over which computation will be performed and defines the functions and relations associated with the symbols of \( \Sigma \) and \( P_U \). We will also use the extension \( L_{\mathcal{O}} \) of \( L \), in which there is a new constant for each element of \( D \). In an abuse of notation we will use \( D \) both for the set of elements of \( \mathcal{O} \) and the set of constants denoting these elements. \( L_{\mathcal{O}} \) is used in the meta-language, whereas programs and queries are formulas of \( L \). \( B_{\mathcal{O}} = \{ p(d_1, \ldots, d_n) \mid p \text{ is } n\text{-ary, } p \in P_U, d_1, \ldots, d_n \in D \} \).

An atom is of the form \( p(t_1, \ldots, t_n) \), where \( p \) is an \( n \)-ary symbol in \( P_U \) and the \( t_i \) are terms. A literal is either an atom or the negation of an atom. Where \( A \) is a literal, \( \text{pred}(A) \) denotes its associated predicate symbol. A primitive constraint is of the form \( p(t_1, \ldots, t_n) \), where \( p \) is an \( n \)-ary symbol in \( P_U \) and the \( t_i \) are terms. A possible constraint is a formula built from primitive constraints using the usual logical connectives and
quantifiers. The allowed constraints (or, simply, constraints) are a (for the moment, unspecified) subset of the possible constraints, which contains all possible equations of terms and is closed under conjunction and existential quantification.

Throughout this paper we will use interchangeably the notions of a conjunction of formulas such as atoms and constraints, and a multiset of the same. We will use $A = H$, where $A$ and $H$ are atoms with the same predicate symbol, to denote the conjunction of equations formed by equating the corresponding arguments of $A$ and $H$.

A $\mathcal{G}$-model (which we will abbreviate to model) for a set of sentences $\Sigma$ is a structure for $L_{\mathcal{G}}$, which extends $\mathcal{G}$ such that the meaning of a constant $d$ is the element $d$ and the structure is a model (in the usual sense) of $\Sigma$. (A structure $\mathcal{G}$ extends a structure $\mathcal{H}$ if they have the same set of elements and every symbol in the language of $\mathcal{H}$ is in the language of $\mathcal{G}$ and is given the same meaning in $\mathcal{G}$.) By $\Sigma \models F$ we denote that $F$ is valid in every $\mathcal{G}$-model of $\Sigma$. A conjunction of constraints $C$ is said to be consistent (or satisfiable) if there are values (from $\mathcal{G}$) for the free variables $y$ such that every constraint is true in $\mathcal{G}$, that is, $\exists y C$. In general, the execution of a program and the application of transformations require tests for consistency. So, practically, it is necessary that consistency be decidable. There are numerous useful such domains, including real arithmetic, linear real arithmetic, complex arithmetic, finite trees with subterm ordering or lexicographic path ordering, rational trees, Boolean algebras, integers modulo $n$, ... However, the results of this paper are independent of any decidability requirement.

A valuation $r$ is a mapping from $V$ to $D$, which extends to map terms to $D$ and $L_{\mathcal{G}}$ to $L_{\mathcal{G}}$, by replacing each free variable $x$ in a formula of $L_{\mathcal{G}}$ with $r(x)$ and evaluating terms and constraints. We sometimes call the result of applying a valuation to a syntactic object a ground instance of that object. Thus, the result of applying a valuation to a term, atom, etc., is called a ground term, ground atom, etc.

A model $M$ will sometimes be represented by the set of ground literals which are true in $M$ (i.e. the diagram of $M$). A partial model is a consistent set of ground literals. $\text{pos}(M)$ denotes the set of atoms in a partial model $M$.

A deductive database or logic program (or simply program) is a collection of rules of the form

$$H \leftarrow C, A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n,$$

where $C$ is an allowed constraint and $H, A_1, \ldots, A_m, B_1, \ldots, B_n$ are atoms ($m \geq 0, n \geq 0$). The positive literals and the negative literals are grouped separately purely for notational convenience. $H$ is called the head of the rule and $C, A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n$ is called the body. If the outermost quantifiers of $C$ in prenex form are existential they may be omitted, provided the corresponding variables occur only in $C$. If $m = n = 0$ then the rule is called a unit rule. A rule can be regarded logically as the sentence

$$\forall x \quad H \lor (C \lor A_1 \lor \cdots \lor A_m \lor B_1 \lor \cdots \lor B_n),$$
where \( x \) is the set of variables in the rule. A ground instance of a rule is the result of applying a valuation for \( x \) to the rule such that \( C \) evaluates to True under this valuation, and then deleting duplicate body atoms.

To simplify the exposition we assume that the rules are in a standard form, where all arguments in atoms are variables and each variable occurs in at most one atom. This involves no loss of generality since a rule \( p(t_1, t_2) \leftarrow C, q(s_1, s_2) \) can be replaced by the equivalent rule \( p(x_1, x_2) \leftarrow x_1 = t_1, x_2 = t_2, y_1 = s_1, y_2 = s_2, C, q(y_1, y_2) \). We also assume that all rules defining the same predicate have the same head and that no two rules have any other variables in common (this is simply a matter of renaming variables).

A rule \( A_1 \leftarrow C_1, B_1 \) rule-subsumes the rule \( A_2 \leftarrow C_2, B_2 \) if there is a substitution \( \theta \) such that \( A_1\theta \equiv A_2 \), \( B_1\theta \equiv B_2 \) and \( \models C_2 \rightarrow C_1\theta \). If two rules rule-subsume each other we say they are subsumption-equivalent. A rule \( r_1 \) rule-subsumes a rule \( r_2 \) iff the ground instances of \( r_2 \) are a subset of the ground instances of \( r_1 \). Rule-subsumption differs from the usual subsumption of clauses since the head atom is distinguished. For example \( p \leftarrow \neg q \) and \( q \leftarrow \neg p \) are identical as clauses, but neither one rule-subsumes the other. However, whenever a rule rule-subsumes another it also subsumes the other in the clausal sense.

A complete logic program (or Clark completion of \( P \)) \([10]\) is a collection \( P^* \) of predicate definitions, each of the form

\[
p(x) \leftarrow \exists y_1(x = t_1 \land B_1) \lor \exists y_2(x = t_2 \land B_2) \lor \cdots \lor \exists y_n(x = t_n \land B_n),
\]

corresponding to the collection of all rules in \( P \) with \( p \) in the heads

\[
p(t_1) \leftarrow B_1,
\]

\[
p(t_2) \leftarrow B_2,
\]

\[
\vdots
\]

\[
p(t_n) \leftarrow B_n,
\]

where \( y_i \) denotes the variables in the \( i \)th rule above, and each \( B_i \) is a (possible empty) conjunction of constraints and literals. If \( p \) does not appear in the head of a rule then \( P^* \) contains \( \neg p(x) \).

### 3. Program dependencies

Program dependencies have played an important role in defining classes of programs (and goals) for which SLDNF resolution is complete with respect to the
Clark-completion semantics [26, 15]. They have also been used to define classes of programs which satisfy model-theoretic properties such as the existence of a perfect model and the consistency of the Clark completion.

In the following definitions we consider the set of rules formed by taking all ground instances of rules in a module or program \( P \). We follow the notation of [15]. \( A \) and \( B \) range over ground atoms.

\( A \not\subseteq_{+1} B \) if \( A \) appears in the head of a ground rule and \( B \) is a positive literal in the body of that rule.

\( A \not\subseteq_{-1} B \) if \( A \) appears in the head of a ground rule and \( \neg B \) is a negative literal in the body of that rule.

A dependency \( A \not\not\subseteq B \) arises from a rule \( r \) if the ground rule used to demonstrate the dependency is an instance of \( r \).

\( A \not\subseteq B \) if \( A \not\subseteq_{+1} B \) or \( A \not\subseteq_{-1} B \).

\( \not\subseteq \) is the transitive reflexive closure of \( \not\subseteq \).

\( A \not\supseteq B \) iff \( A \not\subseteq B \) and \( B \not\subseteq A \).

\( \not\supseteq \) and \( \not\subseteq \) are the least relations such that

\[ A \not\supseteq_{+1} A \]

and

\[ A \not\not\subseteq B \quad \text{and} \quad B \not\not\subseteq C \quad \text{implies} \quad A \not\not\subseteq_{i,j} C, \]

where \( i \cdot j \) denotes multiplication of \( i \) and \( j \). Essentially, \( \not\subseteq_{+1} \) denotes a relation of dependence through an even number of negations and \( \not\subseteq_{-1} \) denotes dependence through an odd number of negations. As is usual, we will write \( A \not\subseteq B \) when \( B \not\subseteq A \) and, similarly, for the other relations.

\( A \not\subseteq_{+1} B \) iff \( A \not\subseteq_{+1} B \) and \( A \not\subseteq_{-1} B \).

\( A \not\not\supseteq B \) iff \( A \not\not\subseteq B \) and not \( A \not\not\subseteq B \).

\( \not\not\subseteq \) denotes the transitive closure of \( \not\subseteq \).

The following facts are easy to verify.

**Proposition 3.1.**

\[ A \not\subseteq B \quad \text{iff} \quad A \not\subseteq_{+1} B \quad \text{or} \quad A \not\subseteq_{-1} B. \]

\[ A \not\subseteq_{+1} B \subseteq_{+1} C \quad \text{implies} \quad A \not\subseteq_{+1,i} C, \]

\[ A \not\subseteq_{-1} B \not\subseteq_{+1} C \subseteq_{-1} D \quad \text{implies} \quad A \not\subseteq D. \]

\[ A \not\subseteq B \not\subseteq_{+1} C \subseteq D \quad \text{implies} \quad A \not\subseteq_{+1} D. \]

\[ A \not\subseteq B \subseteq_{-1} C \subseteq D \quad \text{implies} \quad A \not\subseteq_{-1} D. \]

\[ A \not\subseteq_{+1} B \quad \text{iff} \quad \text{there is a chain} \quad A = X_1 \not\subseteq_{i_1} X_2 \not\subseteq_{i_2} \cdots \not\subseteq_{i_{n-1}} X_n = B \]

and \( i_1 \cdot i_2 \cdots i_{n-1} = j \).
The notion of stratified programs was developed in [8, 2, 29]. In [15] stratification and other properties are expressed in terms of dependencies on predicates. We expand this treatment by applying the definitions to elements of \( B \), in a manner somewhat similar to [18]. A similar generalization has been performed by Cavedon [6, 7]. Any program which is stratified (strict, call-consistent, hierarchical) with regard to predicates will also be stratified (strict, call consistent, hierarchical) in the following more general sense. A program is locally stratified (locally hierarchical) [7] iff it is well-founded and stratified (hierarchical) in the sense below.

Let \( A, B \) range over elements of \( B \). A program \( P \) is

- **stratified** if we never have \( A \equiv B \) and \( A \not\geq -1 B \),
- **strict** if we never have \( A \geq -1 B \) and \( A \not\geq -1 B \),
- **call consistent** if we never have \( A \geq -1 A \),
- **hierarchical** if we never have \( A \equiv B \) and \( A \not\leq B \),
- **well-founded** iff \( <^* \) is well-founded,
- **order-consistent** if \( \leq^* \) is well-founded.

P is not well-founded iff there is a chain \( X_1 \equiv X_2 \equiv \ldots X_n \equiv X_{n+1} \equiv \ldots \), where infinitely many of the \( i_k \)'s are \(-1\) and there is an infinite subsequence \( Y_1, Y_2, \ldots \) of this chain such that, for all \( i \), \( Y_i \not\leq Y_{i+1} \). \( P \) is not order-consistent iff there are chains \( X_1 \equiv X_2 \equiv \ldots X_n \equiv X_{n+1} \equiv \ldots \) and \( Y_1 \equiv Y_2 \equiv Y_3 \equiv \ldots \equiv Y_n \equiv Y_{n+1} \equiv \ldots \), which have a common subsequence \( Z_1, Z_2, \ldots \) and, for each \( k \), the first (second) chain demonstrates that \( Z_k \geq +1 Z_{k+1} \) (\( Z_k \geq -1 Z_{k+1} \)).

Roughly, stratified programs have no recursion through negation, and the law of the excluded middle is inapplicable to strict programs. This notion of call-consistency has been called negative-cycle-free by Sato [24], and order-consistency [24] has been called local call-consistency [7]. Every strict program and every stratified program is call-consistent, and every call-consistent well-founded program is order-consistent.

The model-theoretic results of [15] for definitions with regard to predicates extend, using the above definitions, to all well-founded programs. As one example, if a program \( P \) is call-consistent and well-founded then \( P^* \) is consistent. (This is also a consequence of the stronger result in [24].) Extending the completeness results of [15] for top-down execution requires a definition of **allowed** programs which copes with constraints. A meaningful definition of allowed will depend on the properties of the particular constraint solver and its interaction with the top-down inference engine, and is beyond the scope of this paper. However, when \( \mathcal{G} \) is the Herbrand universe with equations as the only allowed constraints, the completeness results extend to well-founded programs with the usual definition of allowed [15].
A well-ordered partition of \( B \), for a program \( P \) is a transfinite sequence \( \{ H_\alpha : \alpha \geq 0 \} \) of subsets of \( B \), such that

\[
\forall \alpha \neq \beta \implies H_\alpha \cap H_\beta = \emptyset.
\]

\[
\bigcup_{\alpha \geq 0} H_\alpha = B.
\]

A well-ordered partition of \( B \) is a local stratification [18] if for every ground instance

\[
H \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n
\]

of a rule of \( P \); if \( H \in H_\alpha \), then

\[
A_i \in \bigcup_{\beta \geq \alpha} H_\beta \quad \text{for} \quad 1 \leq i \leq m.
\]

\[
B_j \in \bigcup_{\beta \geq \alpha} H_\beta \quad \text{for} \quad 1 \leq j \leq n.
\]

The \( H_\alpha \) are called strata. We use \( P' \) to denote the set of ground instances of rules of \( P \) where the head of the instance is in \( H_\alpha \).

The following proposition gives the relationship between local stratification and stratified. Przymusinski [18] has given a different characterization of local stratification, in terms of a relation which is a well-founded partial order iff \( P \) has a local stratification. The characterization below has the advantage that it decomposes local stratifiability into two independent properties, which can be handled separately. Our proof of correctness of the transformation system benefits from this decomposition.

**Proposition 3.2.** \( P \) has a local stratification iff \( P \) is stratified and well-founded.

**Proof.** Let \( A \ll B \) iff for some \( X, A \ll \ll.X \ll B \). We use the following characterization of local stratifiability, which is a minor variation of characterizations of [18, 6]: A program has a local stratification iff \( \ll \) is well-founded. Suppose \( P \) is stratified and well-founded. Since \( P \) is stratified, \( A \ll B \) iff \( A \ll \ll B \). Hence, if \( A \ll B \) then \( A \ll^* B \). Since \( \ll^* \) is well-founded, so is \( \ll \).

Conversely, if \( \ll \) is well-founded then we cannot have \( A \ll \ll A \). That is, we cannot have \( A \ll \ll X \) and \( X \approx A \). Thus, \( P \) is stratified. Note that if \( A \ll B \) then \( A \ll^\circ B \). Since \( \ll \) is well-founded, so is \( \ll^* \). \( \square \)

Perfect models are defined in [18] for disjunctive deductive databases. We present the definition for nondisjunctive DDBs. First, relations \( \ll_p \) and \( <_p \) on \( B \) are defined to be the smallest relations satisfying

\[
A <_p B \quad \text{if} \quad A \supseteq B.
\]

\[
A \ll_p B \quad \text{if} \quad A \supseteq B.
\]

\[
A \ll_p C \quad \text{if} \quad A \ll_p B \quad \text{and} \quad B \ll_p C.
\]
\[ A \prec_p C \text{ if } A \preceq_p B \text{ and } B \prec_p C, \]
\[ A \prec_p C \text{ if } A \prec_p B \text{ and } B \preceq_p C, \]
\[ A \preceq_p B \text{ if } A \prec_p B. \]

Extending these relations to models of \( P, N \preccurlyeq M \) iff for every ground atom \( A \) true in \( N \) but not in \( M \) there is a ground atom \( B \) true in \( M \) but not in \( N \) such that \( A \prec_p B. \)

A model of \( P \) is perfect if it is minimal with respect to \( \preccurlyeq \).

It is shown in [18] that locally stratified programs over the Herbrand domain have a unique perfect Herbrand model. This can be constructed in a way similar to the standard model of [2], that is, the model is constructed by transfinite induction on the stratification, producing at each step a partial model which is eventually extended to the perfect model. This result extends straightforwardly to other domains. We first give a logical characterization of the atoms added to the partial model in one step. The proof is straightforward. We use the notation \( M |_s = M \cap H_s \) and \( M |_{<s} = M \cap \bigcup_{\beta < s} H_\beta \).

**Lemma 3.3.** Let \( P \) be a locally stratified program with perfect model \( M. \) Let \( A \in H_s. \)

Then \( A \in M \) iff \( P^* \cup M |_{<s} \models A. \)

Two programs are stratification-compatible if there is a well-ordered partition of \( B_j \)
which forms a local stratification for both the programs. Equivalently, \( P_1 \) and \( P_2 \) are stratification compatible if \( P_1 \cup P_2 \) has a local stratification.

The following theorem (and its generalization in Theorem 3.6) provides a powerful

tool for determining whether a transformation preserves the perfect model semantics of a program. It shows that if a program \( P_1 \) is transformed to a stratification-compatible program \( P_2, \) and the transformation preserves a semantics defined by \( S_i, \) then the transformation preserves perfect model semantics.

**Theorem 3.4.** Let \( P_1 \) and \( P_2 \) be stratification-compatible logic programs, and let \( M_1 \) and \( M_2 \) be their corresponding perfect models over the domain \( \Omega. \) Let \( S_1 \) and \( S_2 \) be sentences such that \( M_1 \models S_1 \) and \( \models S_1 \rightarrow P_1. \) If \( \models S_1 \rightarrow S_2 \) then \( M_1 = M_2. \)

**Proof.** Suppose \( M_1 \neq M_2. \) Let \( H_s \) be the least stratum in a common stratification of \( P_1 \) and \( P_2 \) where \( M_1 \) and \( M_2 \) differ, and let \( M \models M_1 |_{<s} = M_2 |_{<s}. \) Let \( A \in H_s. \) If \( A \in M_1 \) then \( P_1^* \cup M \models A \) by Lemma 3.3. Since \( P_1^* \) is a consequence of \( P_1 \) and \( M, \) and \( P_1 \) is implied by \( S_1 \) we must have \( S_1 \cup M \models A. \) Thus, from the hypothesis, \( S_2 \cup M \models A. \) Since \( M_2 \) is a model of \( S_2 \) which extends \( M, \) we must have \( A \in M_2. \) But then, by symmetry, \( M_1 |_{<s} = M_2 |_{<s}, \) which contradicts the initial supposition. \( \square \)

In particular, the theorem applies to the program treated as a logic formula and the Clark completion of the program.
Corollary 3.5. Let $P_1$ and $P_2$ be stratification-compatible logic programs, and let $M_1$ and $M_2$ be their corresponding perfect models over the domain $D$.

(a) If $P_1 \leftarrow P_2$ then $M_1 = M_2$.

(b) If $P_1^* \leftarrow P_2^*$ then $M_1 = M_2$.

In Theorem 3.4 the sentences $S_i$ provide a syntactic characterization of sets of models of programs $P_i$. With slight modifications to the proof we can obtain the following generalization of Theorem 3.4. Let a (model-theoretic) semantics be a mapping $S$ from programs $P$ to sets of models of $P$. We say that one semantics $S_1$ is coarser than another $S_2$ if, for every program $Q$, $S_1(Q) \supseteq S_2(Q)$.

Theorem 3.6. Let $S$ be a semantics coarser than the perfect model semantics. Let $P_1$ and $P_2$ be stratification-compatible. If $S(P_1) = S(P_2)$ then $P_1$ and $P_2$ have the same perfect model.

If we take $S(P)$ to be the set of minimal models of $P$ then we obtain a formal proof of the fact that once the minimal models are fixed it is the form of the program, reflected in the stratification (or, more precisely, the dependency relations), which determines which minimal model is the perfect model. However, it is slightly misleading to view the stratification as choosing a model from among the minimal models. As the theorem shows, any set of models which contains the perfect model could play the same role as the minimal models.

4. Modules

With the prospect of increasingly large and complex deductive database systems comes the problem of managing the many predicate definitions. A module system is a fundamental tool in handling this problem. Also note that a module system in a deductive database can be the basis of an implementation of privacy restrictions. For example, different modules can provide different views of the underlying database, reflecting different privacy restrictions.

A module $P$ consists of predicate definitions and three disjoint sets of predicate symbols, which together include the predicate symbols occurring in the predicate definitions: the set $Exp(P)$ of those predicate symbols defined in $P$ which are accessible outside $P$ (the exported predicates), the set $Imp(P)$ of those predicate symbols used in $P$ which are defined externally to $P$ (the imported predicates), and the set of predicate symbols which are purely internal to $P$ (the local predicates). Module composition associates exported predicates in some modules with imported predicates in another module. We do not discuss any particular syntax for expressing modules and their composition. However, we do make some assumptions about the semantics and the use of the module system.
We assume that a module cannot in any way modify the sets of function symbols and constraints which may be used in the module. The domain \( \mathcal{D} \) and the allowed constraints must be the same for every module. Although there may be great advantages in, for example, localizing the use of a function symbol to a single module, this would introduce major complications to the semantics of modules and module composition.

We assume that each predicate is defined within a single module. If a predicate symbol \( p \) has rules defining it in two different modules then two different predicates are defined, and which predicate is associated with a use of the symbol \( p \) depends on the context of the use and the semantics of the module system. This assumption ensures locality properties: when a predicate definition is to be modified, only one module is directly involved, and when a module is modified, only those modules which depend on that module through module composition can be affected by the change.

We assume a hierarchical calling pattern for modules. That is, a module may not import predicates through module composition which are defined in that module, nor may it import predicates from modules which depend on the module. This ensures that all recursion occurs within modules, and not between them.

These assumptions have several useful consequences for logic programs and deductive databases with negation. If every module of a program is well-founded then so is the program. Similarly, if every module is stratified, hierarchical or order-consistent then so is the entire program. Strictness may be violated by the program although every module is strict. For example, if one module contains

\[ p \leftarrow q, r \]

and is composed with another module containing

\[ q \leftarrow s, t. \]
\[ r \leftarrow \neg s \]

then the resulting program is not strict. (Here we use identical predicate symbols to denote the association by module composition of predicate definition and predicate use.) However, the weaker notion of call consistency is preserved under module composition. Essentially, this is because call consistency requires strictness only between mutually dependent predicates and, under the regime of a hierarchical calling pattern, all mutual dependencies must occur within modules.

The perfect model semantics of a module \( P \) with a local stratification is defined as follows. The semantics of \( P \) is a mapping \( \mu_p \) from relations for \( \text{Imp}(P) \) to relations for \( \text{Exp}(P) \), such that if \( Q \) is the set of ground atoms in the relations for \( \text{Imp}(P) \) then the relations for \( \text{Exp}(P) \) are given by the perfect model of \( P \cup Q \), restricted to exported predicates. When there are no imported predicates and every predicate is exported, this semantics is isomorphic to the perfect model semantics of a program. We write \( P \sim P' \) iff \( \mu_p = \mu_{p'} \).
We can also consider an equivalence of modules based on completions. Suppose that modules $P_1$ and $P_2$ have the same imported and exported predicates, but disjoint collections of local predicate symbols. Let $D_1$ and $D_2$ be the definitions of the local predicates of the respective modules $P_1$ and $P_2$. Then define $P_1 \simeq P_2$ if $\models P_1 \equiv D_1 \land P_2 \equiv D_2$. If $P_1$ and $P_2$ are stratification-compatible then so also are $P_1, D_1, Q$ and $P_2, D_2, Q$ for any relation $Q$ for imported predicates. Using Theorem 3.6 we obtain the following theorem.

**Theorem 4.1.** If $P_1$ and $P_2$ are stratification-compatible and $P_1 \simeq P_2$ then $P_1 \sim P_2$.

It is perhaps worth observing that the partial converses of this statement do not hold. We leave the verification of this fact to the reader. The theorem generalizes to any semantics $S$ coarser than perfect model semantics by taking $P_1 \simeq P_2$ to mean $\mu^S_{P_1} = \mu^S_{P_2}$, where $\mu^S_{P_1}(Q) = S(P \cup Q)|_{Exp(P)}$, the semantics of $P \cup Q$ restricted to the exported predicates.

To avoid choosing a particular syntax, we represent a network $N$ of composed modules as a directed acyclic graph (dag) with module names at the nodes and an edge from $P_2$ to $P_1$ labeled with $x$ if $P_1$ calls $P_2$ and $x:Imp(P_1) \rightarrow Exp(P_2)$ is the partial function associating exported predicates of $P_2$ with imported predicates of $P_1$. We write $P \sim x \sim Q$ if there is a (directed) path from $Q$ to $P$. This dag has the extra property that if $P$ has incoming edges labeled with $x_1, \ldots, x_n$ then the domains of the $x_i$ are disjoint. We also have $Imp(N)$, the set of imported predicate symbols not in the domain of any $x$, and $Exp(N)$, some subset of the set of all exported predicate symbols. Our previous assumptions are necessary for this representation to make sense. For simplicity, we assume that each predicate symbol occurs in only one module.

The semantics of a module network $N$ with modules $P_1, \ldots, P_n$ is a mapping from relations for $Imp(N)$ to relations for $Exp(N)$ defined inductively on the dag as follows: Given relations $Q$ for $Imp(N)$, the semantics of a minimal (in the dag ordering) module $P_i$ determines relations for $Exp(P_i)$. Let $Q'$ be obtained from $Q$ by adding, as a relation for $p$, a duplicate of the relation for $r$ for each $p$ and $x$ such that $x(p) = r$ and $r \in Exp(P_i)$. Let $N'$ be obtained from $N$ by deleting the node for $P_i$ and the edges from $P_i$. The process repeats for $N'$ and $Q'$. Of the final set of relations, the subset corresponding to $Exp(N)$ is chosen.

Module composition corresponds to merging connected nodes in the dag. Let $P$ be a module with an incoming edge from $Q$ labeled with $x$. We will write this as $P \sim x \sim Q$. Composition along $P \sim x \sim Q$ is allowed only if there is no module $S$ such that $P \sim \beta \sim S \sim x \sim Q$. (This ensures that a hierarchical calling pattern is maintained.) In this case the result of composition is a module $R = P \otimes Q$ with predicate definitions $P \cup x(Q)$, $Exp(R) = Exp(P) \cup Exp(Q)$ and $Imp(R) = Imp(Q) \cup Imp(P) \sim \text{domain}(x)$. Here $x(Q)$ denotes the set of predicate definitions which contains $Q$ and, for every $p$ such that $x(p) = q$, contains a rule $p(x) \sim q(x)$. The semantics of the resulting module $R$ is the same as the semantics of the network consisting only of $P \sim x \sim Q$. For each module $S$, if $P \sim \beta \sim S$ and $Q \sim \gamma \sim S$ then these edges are merged, and become $R \sim (\beta + \gamma) \sim S$. 

---

**M.J. Maher**
Similarly, if $S \leftarrow \beta - P$ and $S \leftarrow \gamma - Q$ the edges are merged to become $S \leftarrow (\beta + \gamma) - R$. The semantics of a network containing $P \leftarrow x - Q$ is the same as the semantics of the network containing $R$, once appropriate edges have been merged (by induction on the calling relation).

Module composition satisfies some important properties. The result of composition of modules can be represented syntactically as a module. Consequently, it is unnecessary to work with composite program structures; modules suffice. Compositionality (see the proposition below) justifies performing transformations on one module in isolation, ignoring the context in which it occurs. Independence justifies working with a localized part of a network of modules instead of the entire network.

**Proposition 4.2.** Compositionality: If $P_1 \sim P_2$ and $Q_1 \sim Q_2$ then $P_1 \otimes_s Q_1 \sim P_2 \otimes_s Q_2$.

Independence:

If $P \leftarrow x - Q$, $Q \leftarrow \beta - P$ and $P \leftarrow \gamma - R$ then

$$(P \otimes_s Q) \otimes_{\beta + \gamma} R \sim P \otimes_{\beta + \gamma} (Q \otimes_s R).$$

If $P \leftarrow x - Q$ and $P \leftarrow \beta - R$ and neither $Q \leftarrow \ast R$ nor $R \leftarrow \ast Q$ then

$$(P \otimes_s Q) \otimes_{\beta + \gamma} R \sim (P \otimes_{\beta + \gamma} R) \otimes_s Q.$$ 

**Proof (Sketch).** In each case, it is easily verified that the composite modules have the same imported and exported predicate symbols. For independence, the modules have the same predicate definitions. For compositionality, let $\rho_i(S) = x(\mu_{Q_i}(S)[\text{Imp}(Q_i)])_{\text{Exp}(Q_i)}$ for $i = 1, 2$, where $x$ makes relations for $\text{Imp}(P_i)$ from relations for $\text{Exp}(Q_i)$. Then $\rho_1 = \rho_2$ since $Q_1 \sim Q_2$. Let $R_i = P_i \otimes_s Q_i$ for $i = 1, 2$. Now $\mu_{R_i}(S) = \mu_{P_i}(((\rho_i(S) - S)[\text{Imp}(P_i)])_{\text{Exp}(P_i)})$. Since $\rho_1 = \rho_2$ and $P_1 \sim P_2$, we have $\mu_{R_1} = \mu_{R_2}$, that is, $R_1 \sim R_2$.

5. Transformations for DDBs with constraints

We need some further definitions to express the transformations. A variable renaming is an invertible substitution, that is, a substitution $x$ such that for some substitution $x^{-1}$, $x \cdot x^{-1} = x^{-1} \cdot x = \varepsilon$ (where $\varepsilon$ is the identity substitution). A variant of a syntactic object is the result of applying a variable renaming to that object. By new variant we will refer to a variant which has no variables in common with the current context.

It is convenient for describing the transformations to extend the terminology introduced in [28]. A molecule is an existentially quantified (possibly empty) conjunction of constraints and literals $3x C \land A$. For simplicity, we assume that atoms in $A$ have only variables as arguments and no variable appears twice in $A$. No loss of generality is involved since every molecule is logically equivalent to such a molecule. Two molecules $\exists x_1 C_1 \land A_1$ and $\exists x_2 C_2 \land A_2$ are equal if there is a variable renaming $x$ of the variables $x_1$ to the variables $x_2$ such that $A_1 x \equiv A_2$ and $C_1 x \equiv C_2 x$.

A molecule $\exists x_1 C_1 \land A_1$ is a submolecule of the molecule $\exists x_2 C_2 \land A_2$ if there is a variable renaming $x$ of the variables $x_1$ to a subset of the variables $x_2$ such that $A_1 x \equiv A_2$, $C_2 \rightarrow C_1 x$ and $\text{var}(A_2 - A_1 x) \cap x = \emptyset$. That is, $\exists x_1 C_1 \land A_1$ is a submolecule
of \( \exists x_2 C_2 \land A_2 \) if \( \exists x_2 C_2 \land A_2 \iff \exists Z \land (\exists x_1 C_1 \land A_1 x) \) for some variables \( z \) and some conjunction of constraints and literals \( Z \). In this case \( Z \) is said to be the result of subtracting \( 3x_1 C_1 \land A_1 \) from \( \exists x_2 C_2 \land A_2 \). The submolecule relationship can be viewed as a special case of rule-subsumption where rule heads are empty. Conversely, a rule body can be regarded as a molecule; the rule \( A \leftarrow B \) is equivalent to \( A \leftarrow \exists x B \), where \( x \) is the set of variables in the rule which appear in \( B \) but not in \( A \).

For \( p,q \in \Pi \), we say \( q \) depends on \( p \) if \( A_1 \leq A_2 \) for ground atoms \( A_1,A_2 \), with \( \text{pred}(A_1) = p \) and \( \text{pred}(A_2) = q \).

In a series of module transformations we will denote the initial module by \( P_0 \), and the resultant series of modules is \( P_0, P_1, P_2, \ldots, P_i, \ldots \). The perfect model semantics of \( P_i \) will be denoted by \( \mu_i \). Let \( L \subseteq L' \) be the largest language which only has predicate symbols for imported and exported predicates from \( \Pi \). Every module \( P_i \) has a corresponding language \( L_i \) derived from the function symbols \( \Sigma \) and all predicate symbols in \( P_i \) or \( L' \), so that \( L \subseteq L_i \). To simplify the exposition, we assume that if a predicate symbol \( p \) appears in \( L_i \) but not in \( L_{i+1} \), then \( p \) does not appear in any \( L_j \) for \( j > i \).

5.1. The transformations

We consider the following transformations on a module \( P \).

Constraint replacement

The replacement of a rule

\[ A \leftarrow C, B \]

by the rule

\[ A \leftarrow C', B, \]

where \( C \models C' \). We can also eliminate an equation \( X = Y \) between variables and apply a substitution \( \{X \rightarrow Y\} \) to the rule, provided this leaves the rule in standard form.

Definition

The addition of a set of rules

\[ A_j \leftarrow B_j, \quad j = 1, \ldots, k \]

to \( P \), where \( \{\text{pred}(A_j) : j = 1, \ldots, k\} \) is a set of new predicate symbols, that is, predicate symbols not appearing in the language of \( P \). In the context of a series of transformations \( \text{pred}(A_j) \) must not have appeared in a previous module in the series.

Deletion

The deletion of all rules defining a set \( S \) of predicate symbols such that, for every \( p \in S, p \) does not occur in \( L' \) and every predicate symbol in \( P \) which depends on \( p \) appears in \( S \). Deletion can be seen as an inverse of the Definition transformation.
Removal of subsumed rules

The deletion of a rule which is rule-subsumed by another rule of \( P \). We can also allow the replacement of a rule by the subsumption-equal rule.

Removal of (some) tautologies

The deletion of a rule

\[ A \leftarrow C, B, \]

whose body is tautologously false in one of the following ways:

\[ \models \neg C \]

there are literals \( B_1 \) and \( \neg B_2 \) in \( B \) such that \( \models C \rightarrow (B_1 = B_2) \)

Unfolding

The replacement of a rule (the unfolded rule)

\[ A \leftarrow C, B \]

in \( P \) by the rules

\[ A \leftarrow C \cup C_j \cup \{ B' = H \}, B - \{ B' \} \cup D_j, \quad j = 1, \ldots, m, \]

where \( B' \in B \) is a positive literal, and the rules

\[ H \leftarrow C_j, D_j, \quad j = 1, \ldots, m \]

are new variants of the rules in \( P \) such that \( C \cup C_j \cup \{ B' = H \} \) is consistent. Note that if, for every rule in \( P \), the constraint \( C \cup C_j \cup \{ B' = H \} \) is not consistent then the result of unfolding is to delete the unfolded rule.

We require that the unfolded atom \( B' \) is not imported, that is, \( \text{pred}(B') \) is not an imported predicate symbol. We will sometimes make the restriction that there is no self-unfolding, in other words, that the unfolded rule is never an unfolding rule. Equivalently, for no variable renaming \( \psi \) is \( C \cup C \psi \cup \{ B' = A \psi \} \) consistent.

Folding

The replacement of a collection of rules (the folded rules)

\[ A \leftarrow C_i, B_i, \quad i = 1, \ldots, k \]

in \( P \) by the single rule

\[ A \leftarrow C, H \theta, D, \]

provided (a) there are (new variants of) rules (the folding rules)

\[ H \leftarrow C_i, B'_i, \quad i = 1, \ldots, k \]

in \( P \), (b) \( \theta \) is a variable renaming which maps some variables of \( H \) to \( \text{var}(A, C, D) \), (c) there is a constraint \( C \) and conjunction of literals \( D \) such that \( \exists x_i C_i \theta, B_i \theta \) is a sub-molecule of \( \exists y_i C_i, B_i \) \[ \text{where } x_i \text{ is } \text{var}(C_i \theta, B_i \theta) - \text{var}(H \theta) \text{ and } y_i \text{ is } \text{var}(C_i, B_i) - \text{var}(A) \] for \( i = 1, \ldots, k \) and \( C, D \) is the result of subtracting \( C_i \theta, B_i \theta \) from \( C_i, B_i \) for \( i = 1, \ldots, k \),
and (d) for every rule

\[ H \leftarrow C', B' \]

in \( P \), if \( C \land C' \theta \) is satisfiable then the rule is a folding rule.

We require that no rule is simultaneously a folded rule and a folding rule. (This ensures that we do not destroy a rule by folding it with itself.) It then follows that \( C \land (H \theta = A \psi) \land C \psi \) is not satisfiable, for every renaming \( \psi \). Thus, in the new module there cannot be dependencies \( X \equiv Y \equiv Z \), where \( x \) is \( A \mu \), \( Y \) is \( H \theta \mu \), and both dependencies arise from the resulting rule. We call this the nonreflexive property of folding.

Replacement

A replacement rule takes the form

\[ J \Rightarrow K, \]

where \( J \) and \( K \) are molecules with the same free variables. The application of such a replacement rule to a rule

\[ A \leftarrow B \]

consists of the replacement of a submolecule \( B' \) of \( A \) by \( K \theta \), where \( x \) is \( \text{var}(B) - \text{var}(A) \), \( B' = J \theta \) and \( \theta \) is a variable renaming which acts only on the free variables of \( J \), to obtain

\[ A \leftarrow (B - B') \cup K \theta. \]

Constraint replacement is the special case in which \( J \) and \( K \) contain only constraints. It is legal to apply a replacement rule to such a rule only when no predicate symbol appearing in the replacement rule depends on \( \text{pred}(A) \).

We only allow replacement rules to be applied to \( P_i \) if two conditions are met. The first is that we must have \( P_i^* \models J \Rightarrow K \) for some \( j \leq i \) and \( J \) and \( K \) are in \( L_i \) (\( P_i^* \) is said to validate the replacement rule.) This condition allows the validity of replacement rules to be verified at whichever stage in the process of transformation is convenient, and not only at the first stage or at the current stage, as some transformation systems implicitly require.

Let \( Q \) be the subset of rules in \( P_i \) which define predicates which depend on \( \text{pred}(A) \). The second condition requires that \( Q \) be order-consistent. (This condition will always be satisfied in a locally stratified module.) We say that the Replacement occurs in a conservative context: \( Q \) does not affect the (Clark completion) semantics of \( \text{pred}(A) \).

These transformations are extensions, to handle constraints and negation, of transformations in [16]. Although this transformation system is superficially similar to the transformation system of Tamaki and Sato [27, 28] as extended by Seki [25], these transformation systems are, in fact, quite different. The major difference arises from a difference in the definition of Folding. Here the folding rule is in \( P \) whereas in [28, 25] the folding rule must come from \( P_0 \). One consequence is that folding in this transformation system does not have the same power as folding in [28] (see [16]).
However, a comparison of the two entire systems is not so clear-cut. Secondary differences are that [28, 25] allow only a single folding rule (reversible folding), and place more restrictions on the rules which may be introduced by the Definition transformation.

Gardner and Shepherdson [12] have independently defined a transformation system similar to the one presented here. In particular, the form of Folding is the same. That system also has a more general Replacement transformation.

Before we present the main results of this paper, we give an example of the application of the transformation system. The example comes from different optimizations of the original magic sets method [4] which have appeared in the literature. (The magic sets method produces a “compiled” program which, when executed bottom-up, has many of the goal-oriented advantages of top-down execution of the original program. It was proposed first for Datalog programs [4, 21], but extends to definite logic programs [20] and, with some restrictions, to stratified logic programs [3].) In what follows, we adopt the terminology and notation of [5]. In order not to introduce all the notation and terminology of the magic set method, we will apply the transformations to an example program, adapted from an example of [5]. Nevertheless, the applicability of these transformations is independent of the specific example, and the results of Section 6 provide formal justification of the optimizations.

The initial program \( P \) is the following.

\[
\begin{align*}
sg(X, Y) &\leftarrow flat(X, Y). \\
s(X, Y) &\leftarrow up(X, U), sg(U, V), flat(V, W), \neg sg(W, Z), down(Z, Y).
\end{align*}
\]

The magic set method of [4], assuming a bound/free query to \( sg \) and a left-to-right sideways information passing strategy, produces the following program.

\[
\begin{align*}
sg(X, Y) &\leftarrow \_\_sg(X), flat(X, Y). \\
sg(X, Y) &\leftarrow \_\_sg(X), up(X, U), \_\_sg(U), sg(U, V), flat(V, W), \\
\_\_sg(W), \neg sg(W, Z), down(Z, Y). \\
\_\_sg(U) &\leftarrow \_\_sg(X), up(X, U). \\
\_\_sg(W) &\leftarrow \_\_sg(X), up(X, U), \_\_sg(U), sg(U, V), flat(V, W).
\end{align*}
\]

The program also contains a unit rule for \( \_\_sg \) which is determined by the query. By Unfolding each positive call \( sg(X, Y) \) to \( sg \), replacing the consequent occurrences of \( \_\_sg(X), \_\_sg(X) \) by \( \_\_sg(X) \) (the change preserves subsumption-equality), and then Folding the call to \( sg(X) \), we obtain the following program.

\[
\begin{align*}
sg(X, Y) &\leftarrow \_\_sg(X), flat(X, Y). \\
sg(X, Y) &\leftarrow \_\_sg(X), up(X, U), sg(U, V), flat(V, W), \\
\_\_sg(W), \neg sg(W, Z), down(Z, Y). \\
\_\_sg(U) &\leftarrow \_\_sg(X), up(X, U). \\
\_\_sg(W) &\leftarrow \_\_sg(X), up(X, U), sg(U, V), flat(V, W).
\end{align*}
\]
It should be clear that some computationally redundant calls to \( m\_sg \) have been eliminated. An informal argument for the correctness of this new program was given in [3].

We now define predicates \( supp_2, supp_3, supp_4 \) (in that order). Fold the second rule for \( sg \) and the second rule for \( m\_sg \) using these three predicates, and Fold the first rule for \( m\_sg \) using the definition of \( supp_2 \). The resulting program \( P' \) (below) avoids some reevaluation of expressions by using supplementary predicates \( supp_1 \) to hold intermediate results [22, 5].

\[
\begin{align*}
supp_2(X, U) &\leftarrow m\_sg(X), up(X, U). \\
supp_3(X, V) &\leftarrow supp_2(X, U), sg(U, V). \\
supp_4(X, W) &\leftarrow supp_3(X, V), flat(V, W). \\
sg(X, Y) &\leftarrow m\_sg(X), flat(X, Y). \\
sg(X, Y) &\leftarrow supp_4(X, W), m\_sg(W), \neg sg(W, Z), down(Z, Y). \\
m\_sg(U) &\leftarrow supp_2(X, U). \\
m\_sg(W) &\leftarrow supp_4(X, W). 
\end{align*}
\]

Although we have exhibited these transformations for a particular program, analogous transformations apply to any program generated by the magic set method. Using the results of Section 6, it can be shown that the initial magic sets program and the final program have equivalent perfect model semantics, provided the initial magic sets program is locally stratified. The correctness of a compilation process which immediately produces the program \( P' \) from the program \( P \) then follows immediately from the correctness of the original magic sets method, provided \( P' \) is locally stratified.

6. Preservation theorems

A basic transformation system uses only the transformations defined above. Consequently, a module undergoing transformation is totally isolated from other modules. In this section we discuss some properties which are preserved (i.e. held invariant) by the basic transformation system.

The correctness of the basic transformation system with respect to Clark-completion semantics can be viewed as simply another preservation theorem. Before showing this theorem we need the following lemma, which is adapted from Theorem 3.1 of [24]. It states a condition on Definition transformations (and, indirectly, on Deletion transformations) which ensures that the Clark-completion semantics is preserved.

**Lemma 6.1.** Let \( P_{i+1} \) be obtained from \( P_i \) by a Definition transformation where the set of introduced rules is order-consistent.

Then every model of \( P_i^* \) can be extended to a model of \( P_{i+1}^* \).
Proposition 6.2. Let $P_i$ be obtained from $P_0$ by the basic transformation system. Suppose that the set of rules added or deleted in any Definition or Deletion transformation is order-consistent, and no rule unfolds itself. Then

$$P_i^* \models f \iff P_0^* \models f$$

for every formula $f$ expressible in $L'$.

Proof. The proof is by induction on $i$, where the induction hypothesis consists of the consequent of the theorem and the justification of the replacement rules:

$$P_i^* \models f \iff P_0^* \models f$$

for every formula $f$ expressible in $L'$, and if $J \leftrightarrow K$ is in $L_i$ and if

$$P_j^* \models J \leftrightarrow K$$

for some $j < i$ then

$$P_i^* \models J \leftrightarrow K.$$

The base step, $i=0$, is trivial. The induction step proceeds by cases, one for each transformation in the system. For Removal transformations the equivalence of $P_i^*$ and $P_i^*$ is straightforward. For Definition and Deletion transformations, it follows from Lemma 6.1 that

$$P_i^* \models f \iff P_i^* \models f$$

for every formula $f$ expressible in $L_i \cap L_{i+1}$. Consequently, the induction step holds in this case. For Unfolding and Folding transformations, since the unfolding (folding) rules are not themselves unfolded (folded), the completion of these rules is in both $P_i^*$ and $P_i^*$. Consequently,

$$\models P_i^* \leftrightarrow P_i^*.$$

For a Replacement transformation the second part of the induction hypothesis is needed to show that the replacement rule is validated by $P_i$. Let $R = P_i - Q$, where $Q$ is the set of rules of $P_i$ for predicates which depend on the predicate in the head of the rule to which Replacement is applied. $R$ is also a subset of $P_{i+1}$ and, by legality and the conservative context condition, $P_{i+1} - R$ is order-consistent. By Lemma 6.1, every model of $R^*$ can be extended both to a model of $P_i^*$ and to a model of $P_i^*$. It follows that the replacement rule is also validated by $P_i^*$. Using legality, we thus have

$$\models P_i^* \leftrightarrow P_i^*.$$

For the remaining transformations,

$$\models P_i^* \leftrightarrow P_i^*$$

and, so, the induction step holds. $\square$
By a slight adaptation of this proof we obtain the following theorem.

**Theorem 6.3.** Let $P_i$ be obtained from $P_0$ by the basic transformation system. Suppose that the set of rules added or deleted in any Definition or Deletion transformation is order-consistent, and no rule unfolds itself. Then $P_i \simeq P_0$.

If we allow a rule to unfold itself then the previous two results do not hold, in general [16]. Consider the program over the Herbrand universe consisting of the single rule $p(X) \leftarrow p(f(X))$. After unfolding we obtain $p(X) \leftarrow p(f(f(X)))$, and the two Clark completions are not equivalent. However, self-unfolding does preserve the perfect model semantics of locally stratified modules.

**Lemma 6.4.** If $P_i$ is locally stratified and $P_{i+1}$ is obtained from $P_i$ by Unfolding, $m_{i+1} = m_i$.

**Proof.** Let $Q$ be relations for the imported predicates, and let $M_i$ ($M_{i+1}$) denote the perfect model of $P_i \cup Q$ ($P_{i+1} \cup Q$). We show, by induction on the stratification, that $M_i = M_{i+1}$. Suppose (the induction hypothesis) that $M = M_i |_{<s} = M_{i+1} |_{<s}$. Clearly, $P_i \rightarrow P_{i+1}$. Using Lemma 3.3, it follows that $\text{pos}(M_i |_{<s}) \supseteq \text{pos}(M_{i+1} |_{<s})$. To show that $\text{pos}(M_i |_{<s}) \subseteq \text{pos}(M_{i+1} |_{<s})$, we prove by induction that $T_i | n \subseteq T_{i+1} | n$ for every $n$, where $T_i | 0 = M$ and $T_i | (n+1) = T(T_i | n)$ for any function $T$, and $T_i(I) = I \cup \{ A : A \leftarrow B \}$ is a ground instance of a rule of $P_i$, $I \models B_i$ (and, similarly, for $T_{i+1}$ and $P_{i+1}$). Clearly, this holds for $n = 0$.

Let $A \in T_i | (n+1)$. Then there is an instance $A \leftarrow B$ of a rule of $P_i$ such that $B \subseteq T_i | n$. If that rule also appears in $P_{i+1}$ then $A \in T_{i+1} | (n+1)$. Otherwise, that rule is the unfolded rule. Say, $C \in B$ is the instance of the atom at which the unfolding occurred. Then there is an instance $C \leftarrow D$ of a rule of $P_i$ such that $D \subseteq T_i | (n-1)$. Clearly, $A \leftarrow (B - \{ C \}) \cup D$ is an instance of a rule of $P_{i+1}$. Furthermore, $(B - \{ C \}) \cup D \subseteq T_{i+1} | n \cup T_i | (n+1) = T_{i+1} | n \subseteq T_{i+1} | n$ by the induction hypothesis. Thus, $A \in T_{i+1} | (n+1)$. \[ \square \]

We now turn to dependency-related properties. It is clear that Deletions, Removal of subsumed rules and tautologies, and Unfolding all reduce the dependencies in a module and add none. Hence, these transformations preserve call-consistency, stratifiedness, well-foundedness, ... Subsumption-equal rules have the same ground instances; thus, replacement of a rule by a subsumption-equal rule preserves the dependencies. Definition transformations add dependencies. However, such transformations will preserve each of the above properties except strictness, provided the new rules themselves satisfy the property. This is because the new rules, in essence, form a module which calls the module composed of the old rules. Preservation of strictness can require an examination of the entire module, as pointed out in the section on modules.
For Folding and Replacement transformations preservation of these properties is not so obvious. We deal with Folding first.

**Proposition 6.5.** Suppose $P_{i+1}$ is obtained from $P_i$ by folding the rules

$$A \leftarrow C_i, B_i, \quad i = 1, \ldots, k$$

by the rules

$$H \leftarrow C_i', B_i', \quad i = 1, \ldots, k$$

to obtain

$$A \leftarrow C, H \theta, B.$$

Then

(a) If $P_i$ is stratified then $P_{i+1}$ is stratified.

(b) If $P_i$ is call-consistent then $P_{i+1}$ is call-consistent.

(c) Suppose at least one folding rule is not a unit rule. If $P_i$ is strict then $P_{i+1}$ is strict.

(d) If $P_i$ is hierarchical then $P_{i+1}$ is hierarchical.

(e) If $P_i$ is well-founded then $P_{i+1}$ is well-founded.

(f) If $P_i$ is order-consistent then $P_{i+1}$ is order-consistent.

(g) If $P_i$ has a local stratification then $P_i$ and $P_{i+1}$ are stratification-compatible.

**Proof.** In each case we prove the contrapositive. When we refer to a dependency of the form $A \mu \supseteq 1, H \theta \mu$, we will assume that it can only arise from the rule which results from folding, since otherwise the argument is trivial.

We will use the following remark several times in the proofs below: If $X \supseteq i_1, X' \supseteq i_2, \ldots \supseteq i_\nu, Y$ in $P_{i+1}$ then either the same relationships are true in $P_i$ or the chain has as subchain $A \mu \supseteq 1, H \theta \mu$ for some valuation $\mu$. If all dependencies of the form $A \mu \supseteq 1, H \theta \mu$ occur before the last dependency then there is a subchain $A \mu \supseteq 1, H \theta \mu \supseteq F \theta \mu$ for some valuation $\mu$, where $F$ occurs in the body of a folding rule. Consequently, the **similar** chain, obtained by repeatedly replacing subchains $A \mu \supseteq 1, H \theta \mu \supseteq F \theta \mu$ by $A \mu \supseteq F \theta \mu$, holds for $P_i$.

(a) If $P_{i+1}$ is not stratified then the cycle $X \supseteq i_1, X' \supseteq i_2, \ldots \supseteq i_\nu, Y$ holds in $P_{i+1}$ for some $X \in B_y$ and some $i_\nu$ is $-1$. Note that if the cycle contains $A \mu \supseteq 1, H \theta \mu$ for some valuation $\mu$ then the cycle has a subchain $A \mu \supseteq 1, H \theta \mu \supseteq F \theta \mu$, where $F$ occurs in the body of a folding rule. By the remark above, a similar cycle holds in $P_i$ and, so, $P_i$ was not stratified.

(b) If $P_{i+1}$ is not call-consistent then $X \supseteq i_1, X' \supseteq i_2, \ldots \supseteq i_\nu, X$ holds in $P_{i+1}$ for some $X \in B_y$, where $i_1 \cdot i_2 \cdot \ldots \cdot i_\nu = -1$. Following the same argument as the previous part a similar cycle holds in $P_i$ and, so, $P_i$ was not call-consistent.

(c) If $P_{i+1}$ is not strict then $X \supseteq i_1, X' \supseteq i_2, \ldots \supseteq i_\nu, Y$ and $X \supseteq j_1, Y' \supseteq j_2, \ldots \supseteq j_\mu, Y$ hold in $P_{i+1}$ for some $X, Y \in B_y$ and some $i_1, \ldots, i_\nu$ and $j_1, \ldots, j_\mu$, where $i_1 \cdot i_2 \cdot \ldots \cdot i_\nu = +1$ and $j_1 \cdot j_2 \cdot \ldots \cdot j_\mu = -1$. Suppose at least one folding rule is not a unit rule. If the last
dependency of a chain has the form $A \mu \sqsupset_+, H \theta \mu$, then we can extend the chains by $H \theta \mu \sqsupset_j F \theta \mu$, where $F$ occurs in the body of some folding rule. Consequently, the remarks above are applicable. Now by an argument similar to the previous two parts there are similar chains which hold in $P_i$ and so, $P_i$ was not strict.

(d) If $P_{i+1}$ is not hierarchical then $X \sqsupset_+, X \sqsupset_2, \ldots X \sqsupset_+ X$ holds in $P_{i+1}$ for some $X \in B$. Following the same argument as (a) and (b), a similar cycle holds in $P_i$ and so, $P_i$ was not hierarchical.

(e) If $P_{i+1}$ is not well-founded then $X \sqsupset_+, X \sqsupset_2, \ldots X \sqsupset_+ X$ holds in $P_{i+1}$ for some $X \in B$ where (i) infinitely many of the $i_k$’s are $-1$ and (ii) there is an infinite subsequence $Y_1$, $Y_2$, $\ldots$ of this chain such that, for all $i$, $Y_i \not\sqsupset Y_{i+1}$ holds in $P_{i+1}$. As a result of the nonreflexive property, no $X_k$ can serve simultaneously as dependee and dependor for dependencies arising from the new rule in $P_{i+1}$. Thus, all dependencies in the infinite chain of the form $A \mu \sqsupset_+, H \theta \mu \sqsupset_j F \theta \mu$ for some valuation $\mu$ are independent and can be simultaneously replaced by $A \mu \sqsupset_j F \theta \mu$ to obtain a similar infinite chain which holds in $P_i$. Since only positive dependencies have been omitted, this chain must also have infinitely many negative dependencies.

We now obtain a subsequence $Z_1, Z_2, \ldots$ of the similar infinite chain from the sequence of $Y$’s. The sequence of $Z$’s is obtained by replacing every $Y_i$ of the form $H \theta \mu$ which appears in the chain of $X$’s in the subchain $A \mu \sqsupset_+, H \theta \mu \sqsupset_j F \theta \mu$ by the corresponding $A \mu$, and then deleting repetitions. The result is an infinite subsequence of the similar infinite chain. Furthermore, observe that if $U \not\sqsupset W$ holds in $P_{i+1}$ then $U \not\sqsupset W$ holds in $P_i$, since Folding increases the $\leq$ dependencies. Consequently, condition (ii) is satisfied for the sequence of $Z$’s. Thus, the infinite chain demonstrates that $P_i$ was not well-founded.

(f) If $P_{i+1}$ is not order-consistent then there are chains $X_1 \sqsupset_+, X_2 \sqsupset_2, \ldots X_n \sqsupset_+, \ldots$ and $Y_1 \sqsupset_+, Y_2 \sqsupset_2, \ldots Y_n \sqsupset_+, \ldots$ which have a common subsequence $Z_1, Z_2, \ldots, Z_k, \ldots$ and, for each $k$, the first (second) chain demonstrates that $Z_k \sqsupset_+, Z_{k+1}, (Z_k \sqsupset_+, Z_{k+1})$ in $P_{i+1}$. We can employ the similar chains to show that $P_i$ was not order-consistent, unless all but a finite number of the atoms $Z_k$ have the form $H \theta \mu$. In the latter case, for every $k$ greater than some $k_0$, the chain of $X_i$ contains a dependency $H \theta \mu_k \sqsupset_+, F_k \theta \mu_k$, where $Z_k = H \theta \mu_k, F_k$ occurs in the body of a folding rule and, by the nonreflexive property, $F_k \theta \mu_k$ does not have the form $H \theta \mu$. Now $F_k \theta \mu_k \sqsupset_+, H \theta \mu_k \sqsupset_+, \ldots, H \theta \mu_k \sqsupset_+, H \theta \mu_k \sqsupset_+, \ldots$ so that $F_k \theta \mu_k \sqsupset_+, F_k \theta \mu_k \sqsupset_+, \ldots$. Thus, the sequence of atoms $F_k \theta \mu_k \sqsupset_+, \ldots$ also demonstrates that $P_{i+1}$ is not order-consistent and, as argued above, it follows that $P_i$ was not order-consistent.

(g) By the same arguments as (a) and (e), $P_i \cup P_{i+1}$ is stratified and well-founded. □

Folding with only unit rules can destroy strictness. For example, consider the strict program

\[
\begin{align*}
p(X) & \leftarrow X > 0, \quad q(X) \leftarrow X > 0, \neg r(X), \quad r(X) \leftarrow s(X, Y), \quad p(X).
\end{align*}
\]
By folding the second rule with the first we obtain

\[ q(X) \leftarrow p(X), \neg r(X) \]

in place of the second rule, and now \( q(d) \) depends both positively and negatively on \( p(d) \), for every \( d \in D \).

**Proposition 6.6.** Suppose \( P_{i+1} \) is obtained from \( P_i \) using a replacement rule

\[ J \Rightarrow K \]

on a rule

\[ A \leftarrow B. \]

and suppose the replacement is legal. Then

(a) If \( P_i \) is stratified then \( P_{i+1} \) is stratified.

(b) If \( P_i \) is call-consistent then \( P_{i+1} \) is call-consistent.

(c) If \( P_i \) is hierarchical then \( P_{i+1} \) is hierarchical.

(d) If \( P_i \) is well-founded then \( P_{i+1} \) is well-founded.

(e) If \( P_i \) is order-consistent then \( P_{i+1} \) is order-consistent.

(f) If \( P_i \) has local stratification then \( P_i \) and \( P_{i+1} \) are stratification-compatible.

**Proof.** (a, b) If a cycle \( X \equiv_i X' \equiv_{i_2} \cdots \equiv_{i_n} X \) holds in \( P_{i+1} \) and the cycle contains a dependency \( A \mu \equiv k \beta \mu \) arising from the new rule (where \( k \) occurs in \( K \)) then for some \( k' \) in \( K \) we have \( k' \beta \equiv X_1 \equiv \cdots \equiv X_n \equiv A \mu \) where the \( X_i \) are not of the form \( k \beta \) for any \( k \) in \( K \) and any valuation \( \beta \). The dependencies must arise from the unchanged rules of \( P_{i+1} \) and so, must also hold in \( P_i \). But then \( K \) depends on \( A \), which contradicts the legality of the replacement. Thus, any cycle holding in \( P_{i+1} \) must also hold in \( P_i \). In particular, if \( P_{i+1} \) is unstratified or not call-consistent then so is \( P_i \).

(c) It follows almost immediately from the legality of Replacement, which forces a hierarchical relationship between \( A \) and \( J \) and \( K \).

(d) If \( P_{i+1} \) is not well-founded then \( X_1 \equiv_{i_1} X_2 \equiv_{i_2} \cdots X_n \equiv_{i_n} X_{n+1} \equiv_{i_{n+1}} \cdots \) holds in \( P_{i+1} \) for some \( X_k \in B \), where infinitely many of the \( i_k \)'s are \(-1\) and there is an infinite subsequence \( Y_1, Y_2, \ldots \) of this chain such that, for all \( l \), \( Y_l \equiv Y_{l+1} \) holds in \( P_{i+1} \). If a dependency of the form \( A \mu \equiv k \beta \mu \) arising from the new rule occurs only once, say at \( X_n \), then \( X_n \equiv_{i_n} X_{n+1} \equiv_{i_{n+1}} \cdots \) holds in \( P_i \) and demonstrates that \( P_i \) was not well-founded. If such a dependency occurs more than once then for some \( k' \) in \( K \) and some valuation \( \beta \) we have \( k' \beta \equiv X_1 \equiv \cdots \equiv X_n \equiv A \mu \equiv \cdots \) in \( P_i \), where the \( X_i \) are not of the form \( k \beta \) for any \( k \) in \( K \) and any valuation \( \beta \). This contradicts the legality of the replacement and so it follows that if \( P_{i+1} \) is not well-founded then neither is \( P_i \).

(e) If \( P_{i+1} \) is not order-consistent then there are chains \( X_1 \equiv_{i_1} X_2 \equiv_{i_2} \cdots X_n \equiv_{i_n} X_{n+1} \equiv_{i_{n+1}} \cdots \) and \( Y_1 \equiv_{j_1} Y_2 \equiv_{j_2} \cdots Y_n \equiv_{j_n} Y_{n+1} \equiv_{j_{n+1}} \cdots \) which have a common subsequence \( Z_1, Z_2, \ldots Z_k, \ldots \) and, for each \( k \), the first (second) chain demonstrates that \( Z_k \equiv_{i_k} Z_{k+1} (Z_k \equiv_{i_k} Z_{k+1}) \) in \( P_{i+1} \). As in the argument for part (d), if a dependency of the form \( A \mu \equiv k \beta \mu \) occurs twice or more in a chain then we contradict
the legality of Replacement. Hence, such a dependency occurs at most once in each chain, and we can use the infinite subchains of elements which do not depend on such $k\theta_i$ to demonstrate that $P_i$ was not order-consistent.

(f) Applying the same arguments as in (a) and (d), $P_i \cup P_{i-1}$ is stratified and well-founded. □

The preservation of dependency properties is summarized in the following theorem. It is proved by induction on $i$, using the previous propositions.

**Theorem 6.7.** Let $P_i$ be obtained from $P_0$ by a series of transformations. Suppose that uses of the Definition transformation add subprograms which are stratified (call-consistent, hierarchical, well-founded, order-consistent).

If $P_0$ is stratified (call-consistent, hierarchical, well-founded, order-consistent) then $P_i$ is stratified (call-consistent, hierarchical, well-founded, order-consistent).

Suppose the Definition and Replacement transformations are not used, and Folding always uses at least one nonunit folding rule.

If $P_0$ is strict then $P_i$ is strict.

It is clear that the result for strict modules can be strengthened by allowing Definition and Replacement transformations under some conditions. However, testing these conditions would, in general, require examining the entire module.

It follows immediately from Theorem 6.7 and Proposition 3.2 (and the remarks before it) that, under the conditions of the theorem, the transformation system preserves local stratifiability, local hierarchicality and local strictness.

We now demonstrate the correctness of the basic transformation system with replace to the perfect model semantics for locally stratified modules.

**Theorem 6.8.** Let $P_i$ be obtained from $P_0$ by a series of transformations. Suppose $P_0$ has a local stratification, and that the Definition transformation only introduces subprograms which have a local stratification, and no rule unfolds itself.

Then $P_i$ has a perfect model semantics and $P_i \sim P_0$.

**Proof.** From Theorem 6.7, $P_i$ has a local stratification, for every $i$, and, so, its perfect model semantics $\mu_i$ exists. The remainder of the proof is by induction on $i$. If $P_i$ is obtained by Definition then the hypothesis of this theorem ensures that the hypothesis of Theorem 6.3 is satisfied. If $P_i$ is obtained by Deletion then, since $P_{i-1}$ has a local stratification, the hypothesis of Theorem 6.3 is again satisfied. Thus, whichever transformation was used, $P_{i-1} \simeq P_i$ by Theorem 6.3. Using Propositions 6.5(g) and 6.6(f) and previous remarks, $P_{i-1}$ and $P_i$ are stratification-compatible. Thus, by Theorem 4.1, $\mu_i = \mu_{i-1}$ and, so, by the induction hypothesis, $\mu_i = \mu_0$, that is, $P_i \sim P_0$. □
We can remove the restriction on Unfolding by interleaving the above transformations with self-unfolding. The necessity of treating self-unfolding separately demonstrates a limitation of methods based on Theorem 3.6. Combining Theorem 6.8 with Lemma 6.4 we have the following corollary.

**Corollary 6.9.** Let $P_i$ be obtained from $P_0$ by a series of transformations. Suppose $P_0$ has a local stratification, and that the Definition transformation only introduces subprograms which have a local stratification.

Then $P_i$ has a perfect model semantics and $P_i \sim P_0$.

A comparable result is shown in [25] for a transformation system based on the Tamaki–Sato reversible folding. Gardner and Shepherdson [12] give a result similar to Theorem 6.3 for their transformation system. However, their main interest is in the preservation of the procedural SLDNF semantics.

### 7. Discussion

The basic transformation system treats a module in isolation from any module network in which it appears. The applicability of Theorem 6.8 in this setting follows from the property of compositionality (Proposition 4.2). The transformation system can be extended with module composition and decomposition transformations, which provide the ability to exploit the specific context in which a module appears. In this case we can obtain a result similar to Theorem 6.8 for module networks, using the independence property (Proposition 4.2). However, as would be expected, the semantics of individual modules is not preserved.

The well-founded semantics of DDBs [30] applies to a larger class of DDBs than the locally stratified DDBs. In view of results in [19], which show that DDBs with 2-valued well-founded semantics (the so-called saturated DDBs) can be viewed as satisfying a weak form of stratification, our correctness results might extend to the well-founded semantics for saturated DDBs.

It is not difficult to adapt the proofs of Propositions 6.5(e) and 6.6(d) to show that the transformations also preserve positive-order-consistency [11]. As a consequence of Proposition 6.2 and [11, Theorem 3.2], the basic transformation system preserves the stable model semantics [13] of positive-order-consistent programs.

### Acknowledgment

I thank the referees for their comments.
Note added in proof

It has been pointed out to me by David Kemp that a full synthesis of the magic sets method, with the removal of redundant calls to \( m_{--sg} \) as in Section 5, is performed in [32]. That work uses an extension of the transformation system of [27]. However it only applies to definite programs.

References


