HORNLOG: A GRAPH-BASED INTERPRETER FOR GENERAL HORN CLAUSES

JEAN H. GALLIER AND STAN RAATZ*

▷ This paper presents HORNLOG, a general Horn-clause proof procedure that can be used to interpret logic programs. The system is based on a form of graph rewriting, and on the linear-time algorithm for testing the unsatisfiability of propositional Horn formulae given by Dowling and Gallier [8]. HORNLOG applies to a class of logic programs which is a proper superset of the class of logic programs handled by PROLOG systems. In particular, negative Horn clauses used as assertions and queries consisting of disjunctions of negations of Horn clauses are allowed. This class of logic programs admits answers which are indefinite, in the sense that an answer can consist of a disjunction of substitutions. The method does not use the negation-by-failure semantics [6] in handling these extensions and appears to have an immediate parallel interpretation. ◁

1. INTRODUCTION

HORNLOG, first presented in [13], is an example of an application of a proof procedure different from SLD resolution in an attempt to address a larger subset of first-order logic and to allow indefinite answers. Its logical approach can be summarized as follows: Consider any logic program P consisting of *arbitrary* Horn clauses, and queries of the form $Q = \exists z_1 \ldots \exists z_n (\neg H_1 \vee \cdots \vee \neg H_m)$ where $\{H_1, \ldots, H_m\}$ are Horn clauses whose sets of variables are disjoint, and where $\{z_1, \ldots, z_n\}$ is the union of all these free variables. Observe that $P \supset \exists z_1 \ldots \exists z_n (\neg H_1 \vee \cdots \vee \neg H_m)$ is valid if and only if $P \wedge \forall z_1 \ldots \forall z_n (H_1 \wedge \cdots \wedge H_m)$

Address correspondence to Dr. J. H. Gallier, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Received February 1986; revised July 1986; accepted August 1986.

*At same address as J. H. Gallier.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/87/$3.50

is unsatisfiable. This last formula is equivalent to a conjunction of Horn clauses. If $\{N_1, \ldots, N_k\} \subset P \cup \{H_1, \ldots, H_m\}$ is the subset of negative clauses, and $D \subset P \cup \{H_1, \ldots, H_m\}$ is the subset of definite clauses, then, if the set $P \cup \{H_1, \ldots, H_m\}$ is unsatisfiable, there is a subset $D \cup \{N_i\}$ of this set of Horn clauses which is unsatisfiable for some negative clause N_i. The HORNLOG procedure makes use of this fact and applies to any *arbitrary set* of Horn clauses P and query Q as above, and it may return disjunctive answers when the query Q contains certain forms of negation.

The essence of the method which implements this logical approach is to incrementally build a graph encoding a quantifier-free formula which is a conjunction of substitution instances of clauses in the input logic program P and query Q, and to check for the unsatisfiability of the formula represented by the graph using the linear-time algorithm in Dowling and Gallier [8]. If this check shows the formula encoded in the graph is not yet unsatisfiable, then a node is chosen for expansion, the graph is "rewritten" at the node, and the resulting expanded graph is subsequently checked again for unsatisfiability. This process is an example of the *problem reduction* paradigm [26], and continues until one of the following three cases occurs: (1) the formula encoded by the graph is shown to be unsatisfiable, (2) the graph is not expandable, in which case the formula is refutable, or (3) the formula induces a nonterminating sequence of expansions.

The graph mentioned above describes the logical implications defined by the conjunction of substitution instances of clauses from $P \cup \{\neg Q\}$. Its nodes are labeled by substitution instances of atomic formulae in $P \cup \{\neg Q\}$, plus two special nodes, one for **true** and one for **false**. The edges are labeled with indices of substitution instances of the clauses. The fundamental property of such a graph is that the formula $P \supset Q$ is valid if and only if there is a *pebbling*, a kind of path, from **true** to **false**. The use of graphs in theorem-proving and logic-programming systems has a long history, and we will give a detailed comparison between the data structures used in HORNLOG and those used in a number of other related systems, including the *connection graph* proof procedure [18] and the MESON procedure [23] in particular.

In addition to extending the class of logic programs and queries, this approach to the logic-programming problem has three other advantages. First, the representation of the logic program and query as a graph consisting of atomic formulae as nodes leads to an immediate parallel interpretation. In brief, *any* unexpanded node is available for expansion at *any* time in the graph rewriting process. Subject to synchronization, the graph can be viewed as a form of a dataflow characterization of the unsatisfiability of the input logic program and query. We will comment more on this interpretation after explaining the method, but save the details for a subsequent publication. Second, some forms of negation can be expressed directly, without recourse to negation-by-failure semantics. Third, the extension of logic programs to include any arbitrary set of Horn clauses introduces the possibility that the system can return *indefinite* answers, that is, sets of substitutions, in addition to *definite* answers, substitutions which are singleton sets. As it is not entirely obvious how to interpret an answer which consists of a set of substitutions, nor how logic programs and queries which contain arbitrary Horn clauses can be used, we will give a discussion on these topics before describing the method in detail.

2. MODEL-THEORETIC SEMANTICS OF LOGIC PROGRAMS

The fundamental idea behind logic programming [19, 20] is that a proof or refutation of a logical formula can be viewed as a computation, and that from this process an output can be extracted. The goal in logic programming is not simply to prove a formula valid or unsatisfiable, as is the case in theorem proving, but to actually extract *results* from the proof or refutation of this formula. We presently examine the meaning of this statement more precisely.

The paradigm used in logic programming can be described as follows: Given a first-order formula, or logic program, P, expressing a set of facts and assertions and a first-order query formula Q containing some free variables $z_1, \ldots, z_n$, one wants to know whether the formula $P \supset \exists z_1 \ldots \exists z_n Q$ is valid, and find *explicit* terms $t_1, \ldots, t_n$ such that $P \supset Q[t_1/z_1, \ldots, t_n/z_n]$ is valid. However, even if the formula $P \supset \exists z_1 \ldots \exists z_n Q$ is valid, such terms may not exist, as shown in the following example:

Example 2.1. Let $P = (\neg p(a) \vee \neg p(b))$, and $Q = \neg p(x)$. Then $(\neg p(a) \vee \neg p(b)) \supset \exists x \neg p(x)$ is valid, but there is no term t such that $(\neg p(a) \vee \neg p(b)) \supset \neg p(t)$ is valid.

However, the following result holds.

Theorem 2.2. Consider a first-order language without equality having at least one constant. If P is a formula which is the conjunction of universal sentences of the form $\forall x_1 \ldots \forall x_m B$, where B is quantifier-free, and $\exists z_1 \ldots \exists z_n Q$ is a sentence, with Q quantifier-free, then

$$\models P \supset \exists z_1 \ldots \exists z_n Q$$

iff there is some set of n-tuples of ground terms

$$\{(t_1^1, \ldots, t_n^1), \ldots, (t_1^k, \ldots, t_n^k)\}$$

such that

$$\models P \supset Q[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q[t_1^k/z_1, \ldots, t_n^k/z_n].$$

PROOF. Note that $\models P \supset \exists z_1 \ldots \exists z_n Q$ iff $P \wedge \forall z_1 \ldots \forall z_n \neg Q$ is unsatisfiable. Since all formulae in $H \wedge \forall z_1 \ldots \forall z_n \neg Q$ are universal and prenex, by the Skolem-Herbrand-Gödel theorem [12], $P \wedge \forall z_1 \ldots \forall z_n \neg Q$ is unsatisfiable iff there is some unsatisfiable set $P' \cup \{\neg Q[t_1^1/z_1, \ldots, t_n^1/z_n], \ldots, \neg Q[t_1^k/z_1, \ldots, t_n^k/z_n]\}$ of ground substitution instances of formulae in $P \cup \{\forall z_1 \ldots \forall z_n \neg Q\}$. But then, we have

$$\models P'' \supset Q[t_1^1/z_1, \ldots, t_z^1/z_n] \vee \cdots \vee Q[t_1^k/z_1, \ldots, t_n^k/z_n],$$

where $P'' = P_1 \wedge \cdots \wedge P_k$ for a set $P' = \{P_1, \ldots, P_k\}$ of substitution instances of formulae in P, and, because $\models P \supset P''$, this implies

$$\models P \supset Q[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q[t_1^k/z_1, \ldots, t_n^k/z_n]. \quad \square$$

The model-theoretic semantics given by Theorem 2.2 allows *indefinite* answers, in the sense that the results returned are *disjunctions*. If one wants *definite* answers, that is, answers consisting of tuples of terms, as opposed to sets of tuples, it is necessary to place extra restrictions on the formulae P and Q. The class of *Horn*

formulae is a class of formulae for which singleton answers can be guaranteed, provided that certain conditions described below are met.

Recall that Horn formulae are obtained by restricting the form of the conjuncts, or clauses, in the conjunctive normal form of logical formulae. Specifically, P is a Horn formula if and only if every clause in the conjunctive normal form of P contains at most one positive atomic formula, where an atomic formula is of the form $q(t_1, \ldots, t_n)$, q a predicate symbol, $t_1, \ldots, t_n$ terms from some term algebra. If $A, B_1, \ldots, B_p$ denote atomic formulae, Horn clauses can thus take one of the following three forms:

(1) A,

(2) $\neg B_1 \vee \cdots \vee \neg B_p \vee A$, also written $A :- B_1, \ldots, B_p$,

(3) $\neg B_1 \vee \cdots \vee \neg B_p$, also written $:- B_1, \ldots, B_p$.

Clauses of the form (1) are called *axioms*, clauses of form (1) and (2) *definite* clauses, and clauses of the form (3) *goal*, *query*, or *negative* clauses. It is assumed that distinct Horn clauses are universally quantified. A set of Horn clauses is interpreted as the conjunction of these clauses.

The reason that Horn clauses are attractive for logic programming is contained in the following theorem.

Theorem 2.3. Consider a first-order language without equality having at least one constant. For any (finite) set P of universally quantified Horn clauses, the following properties hold:

(i) For any m ($m \geq 2$) sentences $A_i = \exists y_1^i \ldots \exists y_{p_i}^i B_i$, where each B_i is a conjunction of atomic formulae, if

$$\models P \supset A_1 \vee A_2 \vee \cdots \vee A_m,$$

then for some i, $1 \leq i \leq m$, we have

$$\models P \supset A_i.$$

(ii) For any sentence $\exists z_1 \ldots \exists z_n Q$, where Q is a conjunction of atomic formulae, if

$$\models P \supset \exists z_1 \ldots \exists z_n Q,$$

then there is a n-tuple of ground terms $(t_1, \ldots, t_n)$, such that

$$\models P \supset Q[t_1/z_1, \ldots, t_n/z_n]$$

PROOF. The proof of Theorem 2.3 is technical in nature and appears in Appendix A. We note that a similar result is shown for a higher-order extension of PROLOG in [24]. □

The most interesting consequence of Theorem 2.3 is that it delineates a class of formulae for which it is guaranteed that a proof yields singleton answers: logic programs consisting of definite clauses, and queries consisting of existentially quantified conjunctions of atomic formulae. SLD resolution [2, 21, 22] is a refutation procedure which applies to such sets of Horn clauses.

The following theorem, which can be proved from Theorem 2.3, is also important. Indeed, it is the key to handling more general logic programs including either negative clauses as assertions or disjunctions as queries.

Theorem 2.4. Let P be a set of Horn clauses over a language without equality. Consider the partition of P consisting of the set D of all definite clauses in P, and the set $\{N_1, \ldots, N_k\}$ of negative clauses in P. If P is unsatisfiable, then D contains some atomic formula, $\{N_1, \ldots, N_k\}$ is nonempty, and for some i, $1 \leq i \leq k$, the set $D \cup \{N_i\}$ is unsatisfiable.

PROOF. First, we prove that if $D \cup \{N_1, \ldots, N_k\}$ is unsatisfiable, then D contains some atomic formula and the set $\{N_1, \ldots, N_k\}$ is nonempty. If D does not contain any atomic formulae, then every formula in $D \cup \{N_1, \ldots, N_k\}$ contains some negative literal. Then, $D \cup \{N_1, \ldots, N_k\}$ is satisfied in the one-point structure such that every predicate symbol is interpreted as the constant function **false**. If $\{N_1, \ldots, N_k\} = \varnothing$, then every formula in D contains some (positive) atomic formula. Then, D is satisfied in the one-point structure such that, for every atomic formula of the form $q(t_1, \ldots, t_n)$ in D, q is interpreted as the constant function **true**.

Now, since $\{N_1, \ldots, N_k\} \neq \varnothing$, and since each N_i is a universal formula whose matrix is a disjunction of negative literals, $\neg N_i$ is a formula of the form $\exists y_1 \ldots \exists y_p Q_i$, where Q_i is a conjunction of atomic formulae. But $D \cup \{N_1, \ldots, N_k\}$ is unsatisfiable iff $\models D \supset (\neg N_1 \vee \cdots \vee \neg N_k)$, and by Theorem 2.3(i), we conclude that there is some i, $1 \leq i \leq k$, such that $\models D \supset \neg N_i$, which is equivalent to $D \cup \{N_i\}$ being unsatisfiable. □

As application of Theorem 2.4, we sketch how a more general class of logic programs can be handled. Consider logic programs consisting of a set P of arbitrary Horn clauses and queries of the form

$$\exists z_1 \ldots \exists z_n (\neg H_1 \vee \cdots \vee \neg H_m),$$

where $H_1, \ldots, H_n$ are Horn clauses whose sets of free variables are disjoint, and where $\{z_1, \ldots, z_n\}$ is the union of all these free variables. Observe that

$$P \supset \exists z_1 \ldots \exists z_n (\neg H_1 \vee \cdots \vee \neg H_m)$$

is valid iff

$$P \wedge \forall z_1 \ldots \forall z_n (H_1 \wedge \cdots \wedge H_m)$$

is unsatisfiable. But this last formula is equivalent to a conjunction of Horn clauses. From Theorem 2.4, there is a subset $D \cup \{N_i\}$ of this set of Horn clauses which is unsatisfiable for some negative clause N_i. Since in a refutation of the set $D \cup \{N_i\}$, some of the *definite* clauses in the set $\{H_1, \ldots, H_m\}$ may be used more than once, it is possible to have disjunctive answers. In the next section we give some examples of programming in the class of logic programs outlined above.

3. GENERAL HORN-CLAUSE PROGRAMMING: MOTIVATIONAL EXAMPLES

The phrase "Horn-clause programming" has come to be used synonymously with programming in the language PROLOG, which is not quite accurate, since PROLOG applies to a class of logic programs which consist of only definite clauses

and a single negative clause as query. In order to distinguish between more general logic-programming systems and PROLOG, we will introduce the phrase "general Horn-clause programming" to refer to logic-programming systems which admit arbitrary Horn clauses both in the body of the logic program and the query, and the phrase "definite-clause programming" to refer to the language PROLOG.

HORNLOG is an example of a general Horn-clause programming system. We extend the usual procedural interpretation of logic programs [20] as follows: Negative clauses in the body of the logic program are interpreted as *negative constraints*. That is, $:-B_1,\ldots,B_p$ is interpreted as **false** $:-B_1,\ldots,B_p$, or "not the case that $B_1,\ldots,B_p$ all hold simultaneously". The crucial difference between this interpretation and the negation-by-failure semantics used in PROLOG is that in this interpretation any substitutions computed using substitution instances of negative clauses participate in the construction of the answer substitution.

Queries in HORNLOG are of the form

$$Q = \exists z_1 \ldots \exists z_n (\neg H_1 \vee \cdots \vee \neg H_m)$$

where $\{H_1,\ldots,H_m\}$ are Horn clauses whose sets of variables are disjoint, and $\{z_1,\ldots,z_n\}$ is the union of all these free variables. More explicitly, a query is a disjunction of conjunctions of literals, where each conjunct contains at most one negative literal (and distinct conjuncts have disjoint sets of free variables). Thus the following are all examples of legal forms for queries, where B_i is an atomic formula:

$$Q_1 = \exists z_1 \ldots \exists z_n (\neg B_1 \vee \neg B_2 \vee B_3),$$

$$Q_2 = \exists z_1 \ldots \exists z_n (\neg B_1 \wedge B_2),$$

$$Q_3 = \exists z_1 \ldots \exists z_n (B_1 \vee B_2 \vee \cdots \vee B_m).$$

Let us give some examples of logic programs in HORNLOG to illustrate general Horn-clause programming.

Example 3.1. A music library contains analog recordings of *bach* and *mozart*, and a digital recording of *beethoven*. The following facts are known about recordings in general: digital recordings sound great and a recording cannot both sound great and be analog. Suppose the problem is to find which recordings do not sound great.

This information is easily expressed using general Horn clauses. We note that no ordering is assumed on the clauses in this and subsequent examples. They are simple enough that no ordering is needed, and it will subsequently be argued that HORNLOG is most profitably viewed as a parallel system in which clauses are not ordered.

```
:- soundsGreat(X), analog(X).
digital(beethoven).
analog(bach).
analog(mozart).
soundsGreat(X) :- digital(X).

?- ¬soundsGreat(X).
```

The obvious answers of $X = bach$ and $X = mozart$ are returned by the HORNLOG system.

The negative information contained in Example 3.1 is not so easily expressed in PROLOG. If it is coded either as the fact $not(soundsGreat(X), analog(X))$ or as the rule

$soundsGreat(X)$:-
  $not(analog(X))$,
  $digital(X)$.

with query ?- $not(soundsGreat(X))$, the system responds with failure, as the answer substitutions have been lost by using the negation-by-failure semantics. In fact, the well-known PROLOG convention is to introduce predicates which represent explicitly the negative information, i.e. $soundsPoor(X)$:- $analog(X)$. However, this practice becomes increasingly difficult as the amount of negative information increases.

Example 3.2. Four people, *dave*, *dale*, *peter*, and *jessica*, are involved in a crime. Someone shoots and kills *dave* in the garden. At the time of the crime, it is known that *jessica* was in the house and that *dale* and *peter* were not both in the garden, and it is also assumed that one does not shoot oneself, and that one cannot be both in the house and in the garden. Who is innocent? The above puzzle can be formalized as follows:

:- $shot(X, X)$.
:- $inhouse(X), ingarden(X)$.
:- $ingarden(dale), ingarden(peter)$.
$inhouse(jessica)$.
$suspect(dale)$.
$suspect(peter)$.
$suspect(jessica)$.
$suspect(dave)$.
$ingarden(X)$:- $shot(dave, X)$.
$shot(dave, X)$:- $ingarden(X), suspect(X)$.

?- $\neg shot(dave, X)$.

The HORNLOG system returns the expected substitutions, $X = dave$, $X = jessica$, and the indefinite answer $X = dale \vee X = peter$.

A PROLOG solution to the above example has a very different character that does not reflect the *negative* content of the information. It is also worth pointing out that the last two clauses of this program are mutually recursive, and that the HORNLOG method handles such occurrences without looping. The details of this characteristic will be given after the method is presented. Note also that in Example 3.2, an answer was returned which was a disjunction of substitutions, and indefinite answer. Informally, the logic program of Example 3.2 model-theoretically implies that one of $X = dale$ or $X = peter$ is true, but it is not known specifically which one.

It is possible to have logic programs which return only indefinite answers.

Example 3.3. Consider the following logic program:

$:\text{-}\ chairperson(son(X)), chairperson(daughter(Y)).$
$french(yves).$
$french(pierre).$
$german(fritz).$
$likewine(son(X))\ :\text{-}\ french(X).$
$likewine(daughter(X))\ :\text{-}\ german(X).$

$?\text{-}\ \neg chairperson(Z) \wedge likewine(Z).$

There is no term t such that for the logic program P of Example 3.3,

$$\models P \supset (\neg chairperson(t) \wedge likewine(t))$$

However, both of the following formulas, which represent indefinite answers, require the disjunction of substitution instances to be valid:

$$\models P \supset [\neg chairperson(son(yves)) \wedge likewine(son(yves))] \vee [\neg chairperson(daughter(fritz)) \wedge likewine(daughter(fritz))],$$

$$\models P \supset [\neg chairperson(son(pierre)) \wedge likewine(son(pierre))] \vee [\neg chairperson(daughter(fritz)) \wedge likewine(daughter(fritz))].$$

Finally, consider the following example, a well-known puzzle drawn from the blocks world, which shows some of the limits of this method.

Example 3.4. There are three blocks, labeled a, b, and c. Block a sits atop block b, which sits atop block c. Blocks are also colored either green or blue. The color of a is green, the color of c is blue, and the color of b is unknown. Is there a green block on a blue block?

The answer to this problem is yes, and involves reasoning by cases. Block b must be either green or blue. If b is green, then the answer is yes because b is on c, which is blue. If b is blue, then the answer is yes because a, which is green, is on b. Expressing this problem in first-order logic is straightforward. Let

$$P = \{on(a,b), on(b,c), color(a,green), color(c,blue), color(b,green) \vee color(b,blue)\}.$$

Then the following formula holds:

$$\models P \supset \exists X \exists Y [on(X,Y) \wedge color(X,green) \wedge color(Y,blue)],$$

with substitutions a/X, b/Y, or b/X, c/Y such that

$$\models P \supset [on(a,b) \wedge color(a,green) \wedge color(b,blue)] \vee [on(b,c) \wedge color(b,green) \wedge color(c,blue)].$$

Stating the problem in general Horn clauses is not so straightforward. The problem is that $color(b,green) \vee color(b,blue)$ is not equivalent to any Horn clause.

It contains more than one positive literal. Thus it is necessary to be a little devious in expressing this information:

$on(a,b)$.
$on(b,c)$.
$color(a,green)$.
$color(c,blue)$.

{Devious attempt to state that b is either green or blue,
but in fact, it only says that b cannot be both green and blue
at the same time}

:- $color(b,green),color(b,blue)$.

A first and obvious attempt at posing the problem, ?- $on(X,Y) \wedge color(X,green) \wedge color(Y,blue)$, results in failure, since this query is refutable. Thus we must also be devious in our phrasing of the question:

$$? - [\neg color(X,blue) \wedge on(X,Y) \wedge color(Y,blue)] \vee$$
$$[\neg color(Y,green) \wedge on(X,Y) \wedge color(X,green)].$$

This version of Example 3.4 works, but the query is not very natural. In fact, it asks if there is either a nonblue block X on a blue block Y, or a green block X on a nongreen block Y. Since there are only two colors, the query itself expresses the information that block b can be either green or blue. HORNLOG returns the same substitutions given above.

Before presenting the method, we make the following two observations. It is tempting to suggest that the effect of a negative constraint $N_i = :- B_1, \ldots, B_p$ can be simulated in a definite clause program using SLD resolution as follows: Given a set P of arbitrary Horn clauses, let X be a new literal not occurring in P. Let P' be obtained from P by replacing every negative clause $:- B_1, \ldots, B_p$ in P with $X :- B_1, \ldots, B_p$, and adding $:- X$ as the new goal. It is true that P is unsatisfiable iff P' is unsatisfiable, and since P' only contains definite clauses except for the goal $:- X$, SLD resolution can be applied to test P'.

The above argument is correct, but an important point is missing. The set P is obtained by adding to a logic program the negation of each formula occurring in the query, and in the above method, answer substitutions are *lost*.

It is possible to try to argue that the above method can be refined to take care of this problem. However, a refinement that works in the general case will have to mimic our method. This is because, in order to return the correct answer substitution, it is necessary to keep track carefully of all uses of negations of clauses from the query. This can be demonstrated by the following example.

Example 3.5. Let P be the following set of clauses:

$:- p(a)$.

$:- p(b)$.

$p(X) :- q(X)$.

$? - \neg q(Z)$.

There are two answers: $Z = a$ and $Z = b$. We can form the set P' obtained by adding $q(W)$ and using a new goal $r(Z)$ as follows:

$r(U) :- p(a).$

$r(V) :- p(b).$

$p(X) :- q(X).$

$q(W).$

$? - r(Z).$

P' is unsatisfiable, but unfortunately, the answers are lost. The problem is that there is no way of asserting that U, V, and W are in fact the same variable. Such an assertion would even violate the fact that clauses are universally quantified.

Also, if Q is of the form $Q = \exists z_1 \ldots \exists z_n(\neg H_1 \vee \cdots \vee \neg H_m)$, $m > 1$, for the purposes of showing unsatisfiability, it is equivalent to conjoin an additional clause C to the body of a logic program P, or disjoin the negation of C to the query Q. However, for the purposes of defining the answer substitution there is a difference. Clause C included in the query Q can contribute free variables $\{z_{k_1}, \ldots, z_{k_i}\}$ to the query

$$Q' = \exists z_1 \ldots \exists z_n \exists z_{k_1} \ldots \exists z_{k_i}(\neg H_1 \vee \cdots \vee \neg H_m \vee \neg C)$$

that would not be present if C were in the body of the logic program. If the additional clause C defines Q to be of the form $Q = \neg C$, C clearly cannot be placed in the body of the logic program, since a logic program with a null query returns no answer.

It should be pointed out that the negation-by-failure method [6] is not sound for general Horn clauses. Consider the program

$p(b) :- \neg p(a).$

$:- p(b).$

$? - p(X)$

Using the negation-by-failure strategy, since $p(a)$ cannot be proved, $\neg p(a)$ is true, and clause $p(b) :- \neg p(a)$ yields the truth of $p(b)$. Hence, the answer $[b/X]$ will be returned, saying that $p(b)$ is **true**. However, the clause $:- p(b)$ asserts that $p(b)$ must be **false**.

In the next section we describe the refutation procedure underlying HORNLOG which applies to arbitrary sets of Horn clauses.

4. HORNLOG: A REFUTATION PROCEDURE BASED ON GRAPH REWRITING

The method underlying HORNLOG is inspired by Herbrand's theorem. Its essence consists of incrementally building a graph that encodes a first-order quantifier-free formula, and checking for unsatisfiability of this formula using a linear-time algorithm [8]. If this check for unsatisfiability fails, the graph is rewritten by choosing a node and expanding it, and the expanded graph is again checked for

unsatisfiability. The process terminates if the graph is shown to be unsatisfiable, or if it can no longer be expanded, in which case the query formula is shown to be refutable. The process may also enter into a nonterminating sequence of expansion steps. The algorithm that checks for unsatisfiability is not a resolution method, and has the property that the truth of each node is checked at most once.

This strategy is an instance of the *problem-reduction* paradigm [26]. The method presented here, in the context of a logic-programming interpreter, is new primarily because the underlying data structure, which we call an *H-graph*, is a *graph* and not an AND/OR tree, and because answer substitutions are returned for general Horn-clause programs. Along with representational issues, a major difference between these two data structures is that an AND/OR tree can be shown to be unsatisfiable if its leaves are substitution instances of axioms. This simple property does not necessarily hold in the case of graphs. There is also a subtle but fundamental distinction between methods that mark nodes in an AND/OR tree as "already seen", or "identical to some other node", and we will point out the differences.

The presentation of the method will be divided into four parts: (1) the definition of the underlying H-graph data structure, (2) a description of the procedure for constructing and expanding H-graphs, (3) a description of the algorithm which tests an H-graph for unsatisfiability, and (4) discussion of the main interpreter and related details. In the context of this description, we will use the following notation. Let P be a logic program with query $Q = \exists z_1 \ldots \exists z_n(\neg H_1 \vee \cdots \vee \neg H_m)$, and assume that $P \cup \{H_1, \ldots, H_m\}$ is partitioned into three subsets:

(1) The set $\{N_1, \ldots, N_{n_n}\}$ of negative clauses.

(2) The set $\{F_1, \ldots, F_{n_a}\}$ of axioms.

(3) The set $\{C_1, \ldots, C_{n_d}\}$ of definite clauses of the form $A :- B_1, \ldots, B_p$.

For the sake of clarity, we will present the method in a "theorem proving" form that ensures completeness, and consider the changes necessary to realize a logic-programming interpreter in the conclusion section.

4.1. *Definition of an H-graph*

An H-graph is defined as follows.

Definition 4.1. Let P be a logic program. An *H-graph* G for P is a directed edge-labeled graph denoted by the triple (S, E, L), where S is a set of nodes that are substitution instances of atomic formulae in P, L is a set of labels, each label a pair (C, σ) consisting of a clause C in P and a substitution σ, and E is a subset of $S \times L \times S$ of ordered triples called edges. Each H-graph has two special nodes, called *nodefalse* and *nodetrue*, and all nodes have numerous fields, including a *truth* field, an *age* field, and a *status* field. The truth field of a node is set to **true** if the node is a substitution instance of an axiom, and **false** otherwise. The use of the age and status fields will be discussed later. Given that for any edge $e = (n_1, l, n_2)$, n_1 is the source of e, n_2 is the target of e, and l is the label of e, and that for any node N in G, $\{(C_1, \sigma_1), \ldots, (C_k, \sigma_k)\}$ is the set of labels of all

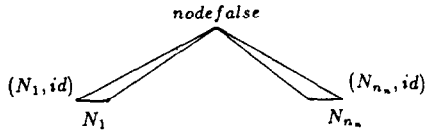

FIGURE 1. Initial H-graph of negative clauses.

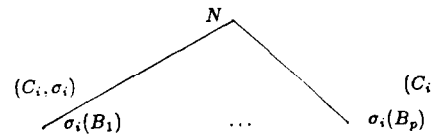

FIGURE 2. Definite clauses.

edges with source N, the sets L and E satisfy the following three provisos:

(i) If C_i is a negative clause of the form $:- B_1, \ldots, B_p$, then $N = nodefalse$, and the target of the jth edge labeled (C_i, σ_i) is $\sigma_i(B_j)$.

(ii) If C_i is a definite clause of the form $A_i :- B_1, \ldots, B_p$, then $\sigma_i(A_i) = N$, and the target of the jth edge labeled (C_i, σ_i) is $\sigma_i(B_j)$.

(iii) If C_i is a clause consisting of a single atomic formula B in S, then $\sigma_i(B) = N$, and there is a single edge with target *nodetrue* labeled (C_i, σ_i).

Proviso (i) for the logic program $P \cup \{\neg Q\}$, with negative clauses $\{N_1, \ldots, N_{n_n}\}$, is illustrated by Figure 1. Every edge from *nodefalse* to an atomic formula B in the negative clause N_i is labeled with (N_i, id), where *id* is the identity substitution. By convention, we will refer to the graph which consists of just the negative clauses as the *initial* graph. Proviso (ii), relating to definite clauses of the form $A :- B_1, \ldots, B_p$, is illustrated by Figure 2. Note that proviso (iii), for definite clauses which are axioms, is the special case in which N has no new successors.

The graph G encodes the conjunction of the clauses of the form $\sigma(C)$ and hence is indeed a kind of *Herbrand expansion*. We will give some examples after a discussion of how an H-graph is constructed.

4.2. Construction and Expansion of H-graphs

The first step in constructing an H-graph is to construct the *initial* H-graph, denoted by G_0, which consists of the node *nodefalse* and all atomic formulae occurring in any negative clause. An H-graph is then expanded in stages, using definite clauses of the form $A :- B_1, \ldots, B_p$ as *rewrite rules*, under one of two protocols. Let $Q = \exists z_1 \ldots \exists z_n Q'$ be a query. For simplicity of notation, we often identify Q and Q', and call $\{z_1, \ldots, z_n\}$ the set of *output variables* in the query Q.

(1) *All-solutions* protocol: The system can return *all* sets of n-tuples of terms (ground or not) $\{(t_1^1, \ldots, t_n^1), \ldots, (t_1^k, \ldots, t_n^k)\}$ such that

$$\models P \supset Q'[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q'[t_1^k/z_1, \ldots, t_n^k/z_n].$$

This case is similar to the assumption in PROLOG that affixing a semicolon ";" after an answer substitution is a request for another answer. The all-solutions protocol in HORNLOG allows the same option.

(2) *Single-solution* protocol: The system returns only *one* substitution. This assumption has no counterpart in PROLOG. The idea is best understood as a simultaneous attempt to explore all possible trials towards unsatisfiability, and to return the first one that succeeds. We will comment more on this protocol after presenting the method.

The two protocols are mutually exclusive. If an H-graph is initially expanded under one protocol, it cannot later in any state be expanded under the other. Note that there is no reason the answer substitution returned under the single-solution protocol cannot be an indefinite answer. Thus, referring back to Example 3.3, two sets of 2-tuples are logically implied by the logic program, the set $\{(son(yves)),(daughter(fritz))\}$ and the set $\{(son(pierre)),daughter(fritz))\}$. Under the all-solutions protocol, both sets could be returned if the user wished, but under the single-solution, only the set that was arrived at first by the HORNLOG interpreter. However both sets result in indefinite answers.

We will present the method first with the simplifying assumption that the node chosen for expansion unifies with only *one* definite clause. This assumption will be relaxed subsequently.

Let G be an H-graph obtained at some stage of expansion, and assume that $X \in G$, the node chosen for expansion, unifies with the head A of a single definite clause C with most general unifier σ. In order to explain how the H-graph G is expanded, we need to define the graphs $\sigma(G)$, G_C, and $\sigma(G)[\sigma(X) \leftarrow \sigma(G_C)]$.

(1) The graph $\sigma(G)$ is defined as follows: First, the substitution σ is applied to all nodes and edges of the graph G, that is, a node labeled with $\sigma_i(N)$ will be relabeled with $\sigma(\sigma_i(N))$, and an edge labeled with (C_j, σ_j) is relabeled with $(C_j, \sigma_j \circ \sigma)$ (where σ_j is applied before σ in $\sigma_j \circ \sigma$). The second step consists in merging any two nodes having the same label. This means any two distinct nodes v_1 and v_2 having the same label L are merged into a single node v labeled L, and that all edges with target v_1 or v_2 now have target v, and that all edges with source v_1 or v_2 now have source v. Hence, in the resulting graph $\sigma(G)$, nodes have distinct labels. Note that composing substitutions is necessary in order to compute the answer substitution at the end.

(2) Let $C = A :- B_1, \ldots, B_p$ be the clause such that X unifies with A. Before determining whether X unifies with the head of C, the variables in C are renamed apart from the variable occurring in the graph G. The graph G_C consists of a root node labeled with A and nodes labeled with the B_i's as immediate successors. Let $\sigma(G_C)$ be the graph obtained by applying the substitution σ to G_C, as defined in (1). Then, the graph $\sigma(G_C)$ is grafted at node $\sigma(X) = \sigma(A)$ in $\sigma(G)$, and nodes having the same label are merged as in (1), obtaining the graph denoted by

$$\sigma(G)[\sigma(X) \leftarrow \sigma(G_C)].$$

In the special case where the definite clause consists of an axiom F, the graph G_C consists of the single node F, and we have $\sigma(G)[\sigma(X) \leftarrow \sigma(G_C)] = \sigma(G)$ with the truth field of node $\sigma(X) = \sigma(F)$ set to **true**. Thus unification with an axiom does not grow new nodes.

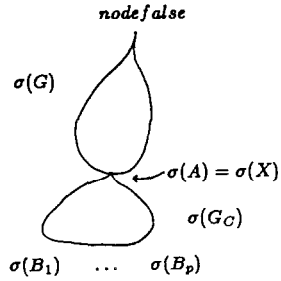

FIGURE 3. Simplified expansion step.

Observe that the process in which the graph $\sigma(G)[\sigma(X) \leftarrow \sigma(G_C)]$ is obtained from the graph G using $A \rightarrow G_C$ $(C = A :- B_1, \ldots, B_p)$ as a "graph rewrite rule" is analogous to a "narrowing step" [9]. However, it is defined for graphs instead of trees. We will write

$$G \underset{C}{\Rightarrow}^{\sigma} G'$$

iff G' is obtained from G in a graph expansion step involving substitution σ and definite clause C. The process is illustrated in Figure 3.

Let us now proceed to the general case, in which a node X chosen for expansion can unify with any number of definite clauses. It is important to point out that the phrase "merging of nodes", as explained previously in (1), is used in its graph-theoretic sense, and implies the *redirection* of arcs. This is not equivalent to marking nodes in an AND/OR tree as, for instance, "already seen" and handling such nodes identically to a previously seen node. Such a strategy does not *propagate* the consequences of the identity of two nodes throughout the graph, and is not equivalent to redirecting the arcs into and out of the node in question. One such consequence, as mentioned in the introduction to Section 4, is that methods to check for the unsatisfiability of a graph and an AND/OR tree are not equivalent. This will be clear when we present the algorithm which performs this check on H-graphs.

As before, let G be an H-graph obtained at some stage of expansion, and node $X \in G$ be chosen for expansion. Let $L = \langle (C_1, \sigma_1), (C_2, \sigma_2), \ldots, (C_k, \sigma_k) \rangle$ be the list of all pairs such that a node X unifies with the head A_j of definite clause C_j with most general unifier σ_j. It is assumed that the variables in the clauses C_j have been renamed apart from the variables occurring in the graph G, and that any two distinct clauses have disjoint sets of variables.

In the general case, it is not necessary to instantiate the entire H-graph G for each substitution σ_j. This can be achieved by attaching an *age* field to every node of an H-graph, as mentioned in Definition 4.1. In order to accomodate the single-solution protocol, the age field is a list of integers, but under the all-solutions protocol, each list contains a single element. The method is that a substitution σ_j is only applied to the subgraph $(G|age(X))$ of G consisting of *nodefalse* and all descendants of *nodefalse* having the same age as the node X currently being expanded. We also maintain a global counter *AgeCounter* which is used in the following way to update *age* fields. Except for the graph $\sigma_1(G|age(X))[\sigma_1(X) \leftarrow \sigma_1(G_{C_1})]$, in which all nodes have the same age as X, for $j = 2, \ldots, k$ the counter *AgeCounter* is

incremented by 1, and the age field of every node in the graph $\sigma_j(G|age(X))[\sigma_j(X) \leftarrow \sigma_j(G_{C_j})]$ is set to *AgeCounter*.

In order to obtain the final graph resulting from an expansion step involving the list of substitutions $L = \langle (C_1, \sigma_1), \ldots, (C_k \sigma_k) \rangle$, certain H-graphs are merged at *nodefalse*. There are two types of merging operations: $\oplus$ under the all-solutions protocol, and $+$ under the single-solution protocol. Given two H-graphs G_1 and G_2, the graph $G_1 \oplus G_2$ is obtained by merging the root nodes of G_1 and G_2, but not performing any other merging. The graph $G_1 + G_2$ is obtained by merging the root nodes and all nodes having the same label. In this case, the age lists of the two nodes being merged are also merged. Let $(G|\overline{age(X)})$ be the subgraph of G consisting of *nodefalse* and all descendants of *nodefalse* whose age is *not equal* to $age(X)$.

Under the all-solutions protocol the graph G' obtained as the result of the expansion step involving the list of substitutions $L = \langle (C_1, \sigma_1), \ldots, (C_k, \sigma_k) \rangle$ is

$$\left(G|\overline{age(X)}\right) \oplus \sigma_1(G|age(X))\left[\sigma_1(X) \leftarrow \sigma_1(G_{C_1})\right] \oplus \cdots$$
$$\oplus \sigma_k(G|age(X))\left[\sigma_k(X) \leftarrow \sigma_k(G_{C_k})\right].$$

Under the single-solution protocol, the graph G' is

$$\left(G|\overline{age(X)}\right) + \sigma_1(G|age(X))\left[\sigma_1(X) \leftarrow \sigma_1(G_{C_1})\right] + \cdots$$
$$+ \sigma_k(G|age(X))\left[\sigma_k(X) \leftarrow \sigma_k(G_{C_k})\right].$$

In the first case, we write

$$G \underset{\oplus\ (C_1,\ldots,C_k)}{\Rightarrow^{(\sigma_1,\ldots,\sigma_k)}} G',$$

and in the second case,

$$G \underset{+\ (C_1,\ldots,C_k)}{\Rightarrow^{(\sigma_1,\ldots,\sigma_k)}} G'.$$

Note that in the all-solutions protocol, merging of nodes only occurs within subgraphs whose nodes have the same age. Under the single-solution protocol, the only condition for merging nodes is that their labels are identical, and nodes having different ages can be merged.

In order to guarantee completeness, it is crucial that the expansion of the H-graph be, in some sense, *fair*, i.e. that no node waits forever for expansion. This corresponds to a breadth-first expansion strategy. To ensure fairness, the *status* field of each node is assigned one of the following four values:

(1) *young*, meaning that this is a newly created node;

(2) *mature*, meaning that the node is ready to be expanded;

(3) *old*, meaning that this node has been expanded;

(4) *dead*, meaning that expansion of this node leads to a *dead end*.

The change of *status* of a node obeys the following fairness rules: (1) A young node can become mature only after all mature nodes have become old, (2) all young nodes must become mature, (3) all mature nodes must become old. When all mature nodes have become old, all young nodes become mature. When a definite clause

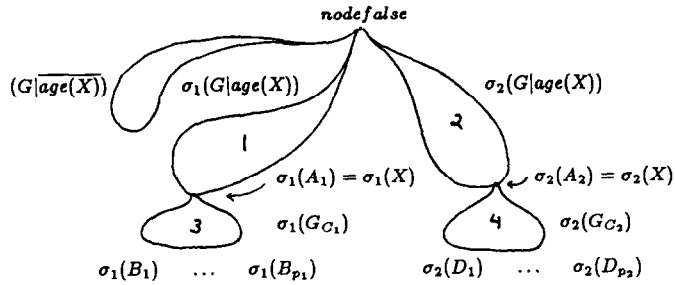

FIGURE 4. Expansion step.

$A :- B_1, \ldots, B_p$ is used to expand an H-graph, the status field of the substitution instance of the head $\sigma(A)$ is set to old, and the status fields of the substitution instances of its successors $\sigma(B_i)$, $1 \leq i \leq p$, are set to young. If the definite clause is an axiom A, then the status field of the substitution instance $\sigma(A)$ is set to old.

We will illustrate the expansion step in the HORNLOG method with three examples, two abstract and one concrete.

Example 4.2. Let P be a logic program partitioned into the set $\{N_1, \ldots, N_{n_n}\}$ of negative clauses, the set $\{C_1, \ldots, C_{n_d}\}$ of definite clauses of the form $A :- B_1, \ldots, B_p$ $(p \geq 1)$, and the set $\{F_1, \ldots, F_{n_a}\}$ of axioms. Figure 4 shows the graph at some expansion stage in which mature note X, chosen for expansion, has unified with the head A_1 of the definite clause $C_1 = A_1 :- B_1, \ldots, B_{p_1}$ with substitution σ_1, and also with A_2, the head of the definite clause $C_2 = A_2 :- D_1, \ldots, D_{p_2}$ with substitution σ_2. The first unification has resulted in graph $\sigma(G|age(X))[\sigma_1(X) \leftarrow \sigma_1(G_{C_1})]$ with the subgraph containing root $\sigma_1(X) = \sigma_1(A_1)$ and immediate descendants $\sigma_1(B_1), \ldots, \sigma_1(B_{p_1})$ being grafted at $\sigma_1(X)$, and the second unification resulted in the graph $\sigma_2(G|age(X))[\sigma_2(X) \leftarrow \sigma_2(G_{C_2})]$. Both $\sigma_1(X)$ and $\sigma_2(X)$ have their status set to old, and all the $\sigma_1(B_i)$ and $\sigma_2(D_j)$ have their status set to young.

Under the all-solutions protocol, merging occurs in region 1, and separately in region 2. After $\sigma_1(G_{C_1})$ and $\sigma_2(G_{C_2})$ have been grafted on, merging occurs again between nodes in regions 1 and 3, and separately, between nodes in regions 2 and 4. Under the single-solution protocol, in addition, merging of nodes occurs across all regions. The generalization to the case in which the node chosen for expansion unifies with any number of definite clauses should be clear.

Example 4.3. Consider the following set of clauses, which are an example of general Horn-clause programming as defined in Section 3:

1. $:\text{-}p(X), q(X,Y), u(Z, f(Z))$.
2. $p(a)$.
3. $n(b)$.
4. $q(X,Y) :\text{-} r(X,Y), s(X,Y)$.
5. $r(X,Y) :\text{-} t(f(X),Y)$.
6. $s(X,Y) :\text{-} t(f(X),Y), m(b)$.

7. $t(f(X),Y)$:-$u(Y,f(X))$.
8. $m(b)$:-$n(b)$.
9. $n(b)$:-$m(b)$.
10. ?-$\neg u(a,Y)$.

Note that the first clause is a negative clause used as an assertion, and that the query contains a negation. It is also worth pointing out that this example contains two clauses, 8 and 9, which are mutually recursive. This example has been chosen with simplicity of presentation very much in mind. In particular, for every expansion step, the node chosen for expansion unifies with only a single definite clause, which implies that all nodes have the same age. Thus, for each expansion step, $0 \leq i \leq p-1$,

$$G_{i+1} = \sigma_{i+1}(G_i | age(X))\left[\sigma_{i+1}(X) \leftarrow \sigma_{i+1}(G_{C_{i+1}})\right].$$

The extent of merging is also restricted. (Example 4.4 examines merging in the more general case.)

The initial graph for this example is shown in Figure 5. Nodes are annotated by the schema $label^{(index,status)}$, where *label* is a substitution instance of an atomic formula, *index* is an integer index of nodes, and *status* is the status field of the node, denoted as *m* for mature, *o* for old, and *y* for young. Edges are annotated by (*clause*, σ), where *clause* is the clause associated with the edge and σ is a substitution. We will adopt the rule that the node chosen for expansion will be the mature node of lowest index.

On the first expansion step, the mature node with lowest index is node 2. The procedure maintains for each node, in a manner analogous to the connection-graph procedure [18], a list of clauses whose head *could* unify with the node. In this case, node 2 can only unify with the axiom $p(a)$ with most general unifier $\sigma_1 = [a/X]$. Rewriting by this step yields an *H*-graph with the same structure as that in Figure 5, except that there is an arc from node 2 to *nodetrue*, and the substitution $[a/X]$ has been applied to the graph. Note also that the status of node 2 has been set to old. On the next expansion step, node 3 is rewritten by clause 4, that is, $q(a, Y)$ unifies with $q(X, Y)$ with most general unifier $\sigma_2 = [a/X, Y/Y]$, and results in Figure 6.

After a few more expansion steps, the *H*-graph shown in Figure 7 is reached. For the sake of clarity, we have dropped the annotations on this graph. The first unsatisfiable *H*-graph constructed by the procedure *expandGraph* is shown in Figure 8. It has resulted from expanding the node labeled $u(Y, f(a))$ in the *H*-graph in Figure 7 with substitution $[a/Y]$. Note that a merging of nodes labeled $u(a, f(a))$ has occurred, but has not resulted in any redirection of arcs from the two nodes. In this case, the "merging" is equivalent to marking the second node as "already seen", as done in various methods that use AND/OR trees. In the more complicated case of Example 4.4, however, the two methods are *not* equivalent.

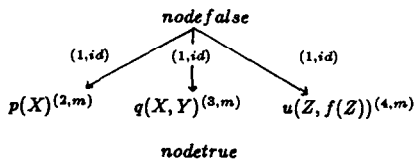

FIGURE 5. Initial *H*-graph for Example 4.3.

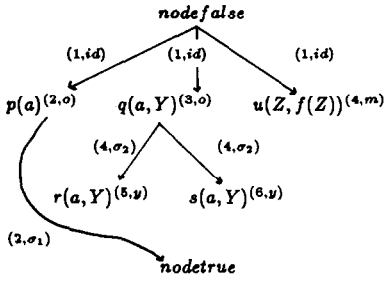

FIGURE 6. Third expansion step in Example 4.3.

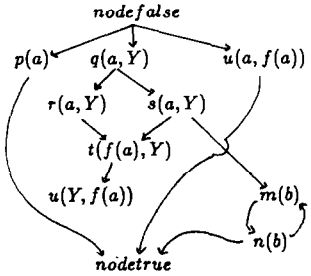

FIGURE 7. Intermediate step in expansion cycle.

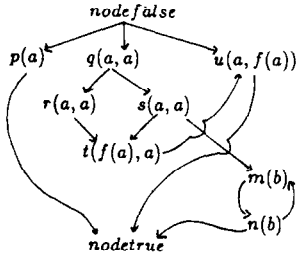

FIGURE 8. First unsatisfiable H-graph for Example 4.3.

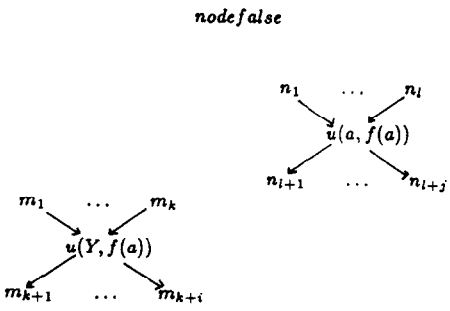

FIGURE 9. Intermediate expansion stage.

Example 4.4. Consider amending the logic program of Example 4.3 so that for some intermediate stage of expansion, an H-graph analogous to that shown in Figure 7 contains the structure illustrated in Figure 9. The node labeled $u(Y, f(a))$ has incoming arcs from nodes labeled $\{m_1, \ldots, m_k\}$, and outgoing arcs to nodes labeled $\{m_{k+1}, \ldots, m_{k+i}\}$. Similarly, the node labeled $u(a, f(a))$ has both incoming and outgoing arcs. Other nodes are not shown. As in Example 4.3, let the next

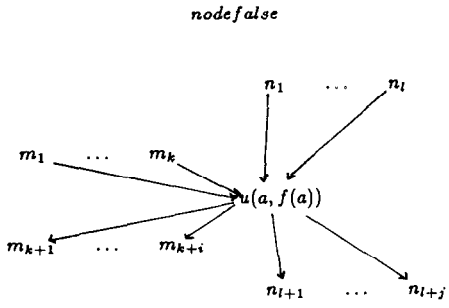

FIGURE 10. Second intermediate expansion stage.

rewriting of this H-graph result from expanding the node labeled $u(Y, f(a))$ with substitution $[a/Y]$, as shown in Figure 10. As before, the two nodes merge, but now *redirection* of arcs occurs, and nodes $\{m_1, \ldots, m_{k+i}\}$ and $\{n_1, \ldots, n_{l+j}\}$ are affected. In this case, the method differs form methods which use AND/OR trees. The generalization to the case in which more than two nodes merge should be clear.

A number of techniques are used in the method to cut down on the size of the graphs. One in particular is important. A mature node becomes *dead* if all unifications fails. The death of a node may cause the death of some of its ancestors according to the following rule: A node X gets the status dead iff for every label (C, σ) of an edge with source X, there is a target node of some edge labeled (C, σ) having status dead. The *death propagation* is conveniently performed using a variation of the same procedure which checks for the unsatisfiability of an H-graph. One more technical detail is that the node *nodetrue* is actually never built. Instead, a node gets value *true* when it is matched with a clause of the form B.

We summarize this presentation in a pseudo-code version of the cental procedure of the expansion step, *expandGraph*. Let P be a logic program with query $Q = \exists z_1 \ldots \exists z_n(\neg H_1 \vee \cdots \vee \neg H_m)$, and again assume that $P \cup \{H_1, \ldots, H_m\}$ is partitioned into the three subsets (1) $\{N_1, \ldots, N_{n_n}\}$ of negative clauses, (2) $\{F_1, \ldots, F_{n_a}\}$ of axioms, and (3) $\{C_1, \ldots, C_{n_d}\}$ of definite clauses of the form $A :- B_1, \ldots, B_p$.

Procedure expandGraph$(G, Flag, Protocol, AgeCounter)$;

{G is of type H-graph, *Flag* is a boolean flag, and *Protocol* is
a boolean indicator of expansion protocol}
begin
repeat
Select any node in G having status *mature*;
Let X be the atomic formula labeling the selected node. Test whether
X is unifiable with any head A of some definite clause
$A :- B_1, \ldots, B_p$ in $\{C_1, \ldots, C_{n_d}\}$, or axiom F_j;
{If X is not unifiable, set status and repeat, i.e., look for another
mature node.}
if all unifications fail **then**
Set the status of X to *dead*;
Delete from G all nodes that are inaccessible as a consequence of
X becoming dead;

else
Let $L = \langle (C_1, \sigma_1), \ldots, (C_i, \sigma_i) \rangle$ be the list of all pairs
such that X is unifiable with the head A_j of a definite clause
C_j in P, with most general unifier σ_j of X and A_j;
{Use each clause C_j in L, which is of the form $A :- B_1, \ldots, B_p$,
$1 \leq j \leq i$, as rewrite rule to expand the H-graph.}
for each pair (C_j, σ_j) in L **do**
rewriteRule$(j,(C_j, \sigma_j),(G|age(X)),G_j, AgeCounter)$
endfor;
if *Protocol* = all-solutions **then**
Set $G = (G|\overline{age(X)}) \oplus G_1 \oplus \cdots \oplus G_i$
else
Set $G = (G|\overline{age(X)}) + G_1 + \cdots + G_i$
endif
endif
until some *mature* node has been selected for expansion, or no *mature*
nodes are left;
if no *mature* node has been selected for expansion **then**
{The graph is thus not unsatisfiable, i.e., the query formula is refutable}
Set *Flag* to **false**
else
{The H-graph G has successfully been expanded and is ready to
be tested for unsatisfiability.}
Set *Flag* to **true**
endif
end;

Procedure rewriteRule$(j,(C, \sigma),G,G_j, AgeCounter)$;

begin
Form a copy $\sigma(G)$ of G by applying σ to every node in G;
Merge nodes having same label in $\sigma(G)$;
if C_j is an axiom **then**
Set the status of $\sigma(X)$ to *old*, and its truth field to **true**
else
Form $G_j = \sigma(G)[\sigma(X) \leftarrow \sigma(G_C)]$, that is,
apply σ to the H-graph consisting of A as a root,
and $B_1, \ldots, B_p$ as immediate successors, obtaining $\sigma(G_C)$,
graft this subgraph at the node labeled with $\sigma(X)$
in the H-graph $\sigma(G)$, and merge nodes having the same label;
Set the status of the node $\sigma(X)$ in G_j to *old*;
Set the status of each node $\sigma(B_l)$, $1 \leq l \leq i$, in G_j to *young*
endif;
{If X has participated in more than one unification, update *AgeCounter*.}
if $j = 1$ **then**
Set the age of each node in G_1 to $age(X)$
else
Increment *AgeCounter*;

Set the age of each node in G_j to *AgeCounter*
endif
end;

We now review the linear-time algorithm which the HORNLOG method uses to check an H-graph for unsatisfiability.

4.3. Checking an H-graph for Unsatisfiability

After each expansion step, the procedure checks the H-graph that has resulted for unsatisfiability, that is, it checks whether the quantifier-free formula that the graph encodes is unsatisfiable and returns the answer substitution if this is the case. This check is performed by interpreting the H-graph as a kind of dataflow graph for propagating truth. Informally, if truth can be propagated to *nodefalse*, the graph is unsatisfiable.

We caution the reader that it is the procedure *expandGraph* which handles variables. The algorithm described in this section which checks an H-graph for unsatisfiability operates on the data structure as a *lexical* object in that it compares the *labels* of nodes and their edges. These labels may contain variables, but as the reader will see in the discussion below, although the algorithm is defined in Dowling and Gallier [8] for the propositional case, it can be used *in conjunction with* the procedure *expandgraph* in the first-order case.

Definition 4.5. Let $G' = (S, E, L)$ be a graph obtained from the H-graph G for some logic program P by *reversing* the edges of G. There is a *pebbling* of a node $M \in S$ from a set $X \subset S$ if either M belongs to X, or for some label (C, σ) corresponding to some clause C in P and substitution σ, there are pebblings of nodes $Y_1, \ldots, Y_n$ from X, where $Y_1, \ldots, Y_n$ are the sources of all incoming edges of M labeled (C, σ).

Thus node M can be pebbled from X if there is a sequence of "pebbling moves" such that starting from nodes in X, a node is pebbled iff for some label (C, σ), all sources of incoming edges labeled (C, σ) are pebbled. The fundamental property about the pebbling of an H-graph is stated in the following theorem, shown in [8].

Theorem 4.6. Let G' be a reversed H-graph for some logic program P. Then G' is satisfiable iff there is no pebbling of nodefalse from nodetrue.

It can be shown that in order to test whether a node M in G' can be pebbled, it is sufficient to visit all nodes reachable from M in the H-graph associated with G'. There are actually several ways of propagating truth in a graph representing a set of clauses. The linear-time algorithm used in HORNLOG is a recursive algorithm which, given any node X, tries to find some label (C, σ) among the set of labels of all edges with source X, such that all the targets of edges labeled (C, σ) can be marked true. Such an algorithm performs a kind of depth-first search. However, there are subtle difficulties in implementing this strategy because a node may have several incoming edges and the H-graph may have cycles; thus a simple scheme of marking the nodes is insufficient.

The solution adopted in HORNLOG is to mark the edges and allow a visit to a node if either (1) there is some unmarked edge to it, or (2) one of its immediate successors has some unmarked edge. This strategy is implemented by including for the nodes and edges in the H-graph various "bookkeeping" fields. For every label $\lambda = (C, \sigma)$ there is a counter *numargs*(λ) which keeps track of the number of target nodes of edges labeled λ whose truth field is **false**. Also, for every node X, there is a list *clauselist*(X), containing the list of all labels (C, σ) such that X belongs to the right-hand side of the clause $\sigma(C)$.

Example 4.7. Consider the graph in Figure 6 for the logic program of Example 4.3. For node 5, *clauselist*$(r(a, Y)) = (4, \sigma_2)$, and for clause 4, *numargs*$(4, \sigma_2) = 2$.

Then, whenever, a node X is marked **true**, the counter *numargs*(C, σ) of every label (C, σ) in *clauselist*(X) is decremented by one, and every node B which is the left-hand side of a clause $\sigma(C)$ such that *numargs*$(C, \sigma) = 0$ is marked **true**.

Example 4.8. We trace the algorithm applied to the last H-graph of Example 4.3, shown in Figure 8. First, $p(a)$, $u(a, f(a))$, and $n(b)$ become **true**, and then truth is pushed all the way up to *nodefalse*. Hence, the above graph is unsatisfiable. Actually, the algorithm *traverse* will traverse the graph and mark the nodes **true** in the following order: visit *nodefalse*, visit $p(a)$, mark $p(a)$ **true**, visit $q(a, a)$, visit $r(a, a)$, visit $t(f(a), a)$, visit $u(a, f(a))$, mark $u(a, f(a))$ **true**, mark $t(f(a), a)$ **true**, mark $r(a, a)$ **true**, visit $s(a, a)$, visit $t(f(a), a)$ already marked **true**, visit $m(b)$, visit $n(b)$, mark $n(b)$ **true**, mark $m(b)$ **true**, mark $s(a, a)$ **true**, mark $q(a, a)$ **true**, visit $u(a, f(a))$ already marked **true**, and mark *nodefalse* **true**.

In addition to the various fields mentioned in the definition of an H-graph, nodes contain a *computed* field, which is set to **true** when the truth field is set to **true** during the check for unsatisfiability in order to avoid recomputing it. The procedure which implements this is given in pseudo-code.

Procedure traverse (*Current*, G);

{*Current is* of type *Node*, G is of type H-graph. *Current.Id*
is read as the *Id* field of *Current*.}
begin
if *Current.Computed* is **false then**
{Call *traverse* recursively. First check to see if *Current* has been
initialized to true in the building of the H-graph, i.e., that it
corresponds to a substitution instance of an axiom.}
if *Current.Truth* is **true then**
Set *Current.Computed* to **true**;
Update the counter for every edge label (C, σ) in *clauselist*(*Current*);
{For every edge label (C, σ), compute the value of the targets of
all edges with source *Current* labeled (C, σ), as long as the
computed field of *Current* is false. A *Successor* of *Current* is
a set of edges with identical labels.}
else
Let *Succ* be the set of *Successors* of *Current*;
for each *Successor* **in** *Succ* **and while** *Current.Truth is* **false do**

```
        {Traverse recursively for each edge labeled (C, σ).}
          for each Edge in Successor do
            {Visit nodes reachable by unmarked edges.}
            Let TargetNode be the target of Edge;
            if Edge.Visited is false then
              Decrement Current.Marked;
              Set Edge.Visited to true;
              traverse(TargetNode,G)
            {If all edges are visited and TargetNode has some unvisited
              outgoing edge, then call traverse}
            else
              if (Current.Marked = 0) and
                (TargetNode.Marked ≠ 0) then
                traverse(TargetNode,G)
              endif
            endif
          endfor
          {If not already computed and all arguments for edge label
            (C, σ) are available, compute the truth value of Current.}
          if Current.Computed is false then
            if numargs((C, σ)) = 0 then
              Update counter for every edge label (C, σ) in ClauseList
                corresponding to Current;
              Set Current.Computed to true;
            endif
          endif
        endfor;
        Set Current.Computed to true;
      endif
    endif
  end;
```

It should be clear now why mutually recursive clauses, as in Example 4.3, do not cause this method to loop. Since nodes are visited only if they are reachable by unmarked edges, or if some successor has an unmarked edge, a nonterminating loop cannot occur.

4.4. The Main HORNLOG *Interpreter*

The basic function of the main interpreter is to interleave graph expansion steps performed by *expandGraph* and checks for unsatisfiability using the procedure *traverse*. In addition, it coordinates the dropping of nodes (under the all-solutions protocol) that are associated with only one substitution instance in order to ensure that the graph contains no redundant information, coordinates the updating of the various fields of nodes and edges, and extracts the answer substitution for an unsatisfiable graph.

Extracting an answer substitution from an H-graph is not as straightforward as it may seem. The answer substitution is not associated with *arbitrary* nodes whose

truth field is **true**, since some nodes may be **true** but not participate in the pebbling of *nodefalse*. In fact, it is necessary to reconstruct this pebbling. This reconstruction is performed by a recursive procedure *findPebbling* that works as follows. Called from a node n labeled **true**, if either n is a leaf or n is *not* the source of a group of edges labeled identically whose targets are all **true**, *findPebbling* visits n, and returns the binding associated with output variables (if any such variables are associated with n). Called from a node n labeled **true** such that n is the source of a group of edges labeled identically whose targets are all **true**, as in the previous case, *findPebbling* visits n and returns the binding associated with output variables, but in addition, *findPebbling* is called recursively for each of the **true** successors in the leftmost group of **true** successors of n.

Example 4.9. After the H-graph of Figure 8 is shown to be unsatisfiable as in Example 4.3, this pebbling is reconstructed, and the binding of the node labeled $u(a, f(a))$ (which is associated with the free variable of the query), $Y = f(a)$, is returned as the answer substitution.

If the graph was expanded under the all-solutions protocol, it contains implicitly different and disjoint trials towards unsatisfiability. The information associated with the present answer substitution is not needed for subsequent trials. However, one has to be very careful not to delete nodes that might be used by subsequent trials because they are the target of several edges. The trick is to first drop edges, and then the nodes which are not accessible from *nodefalse*. Dropping edges is in itself a subtle process. It is incorrect, for instance, given a node X, to drop a group of identically labeled edges with source X even if the target nodes of these edges are all **true**. Such a strategy can delete edges that are not in fact redundant. The correct strategy is to identify a group of edges with source X labeled identically, such that all the target nodes are **true**, and each of these target nodes is *not* the source of a second group of identically labeled edges whose targets are not all **true**. Only in this case can the edges in the true group with source node X be deleted. Also, a node belonging to a pebbling is reset to **false**, unless it has no successors.

During the expansion cycle the case could occur when there are unexpanded nodes with status young, but not more nodes with status mature in the graph. Since one of the fairness rules assumes the presence of a mature node in order to initiate an expansion step, it is necessary to check the H-graph after each expansion step for this situation, and update the status of all young nodes to mature if it does. The procedure *reInitialize* performs this step.

We summarize this information in the following section of pseudo-code:

```
  °
  °
while H-graph G is not unsatisfiable and MoreMatureNodes do
  {Check for unsatisfiability.}
  traverse(G);
  if the truth field of nodefalse is true then
    returnAnswer(G);
    if all-solutions protocol and user wishes another answer then
      {Delete nodes from G that are associated only with present substitution.}
```

```
        dropValidPart(G);
        expandGraph(G, MoreMatureNodes);
        {Update status fields.}
        reInitialize(G);
      else
        return;
      endif
    else
      {The graph is not yet unsatisfiable.}
      expandGraph(G, MoreMatureNodes);
      reInitialize(G);
    endif
  endwhile;
  if G not unsatisfiable then indicate query refutable;
    °
    °
```

We conclude this section with an example showing that the use of the single-solution protocol may prevent finding all substitution answers.

Example 4.10.

$:-p(a).$

$:-p(a), p(e).$

$p(a):-p(c), p(d).$

$?-\neg p(X)$

If we start with the single-solution protocol, since in the initial graph the node labeled with $p(a)$ unifies with the head of two clauses [$p(a):-p(c)$, $p(d)$, and $p(X)$], after expansion and merging, this node becomes **true**, and the answer $[a/X]$ is returned. After *dropValidPart*, this node is reset to **false**, and no nodes are dropped. If the next expansion steps unify $p(e)$, $p(d)$, and $p(c)$ with $p(X)$, in this order, then the disjunctive answer $[e/X] \vee [d/X] \vee [c/X]$ is returned. At this point, *dropValidPart* drops the nodes labeled with $p(c)$, $p(d)$, and $p(e)$, and the procedure stops. The answer $[c/X] \vee [d/X]$ (which is more general than the previous one) is not returned.

The problem is the merging of the two (ground) instances of $p(a)$ with *different ages* in the first expansion step. In order to return all answers, these nodes should not be merged.

5. SOUNDNESS AND COMPLETENESS: A DISCUSSION

In this section we give an argument for the correctness of the HORNLOG method under the all-solutions protocol as a computation procedure, in the sense that its operational semantics agrees with the model-theoretic semantics. The case of the single-solution protocol is similar. Formal proofs of correctness for both protocols can be found in [14, 27].

We will need the following definition.

Definition 5.1. Let P be a logic program with query $Q = \exists z_1 \ldots \exists z_n(\neg H_1 \vee \cdots \vee \neg H_m)$, such that $P \cup \{H_1, \ldots, H_m\}$ is partitioned into subsets $\{N_1, \ldots, N_{n_n}\}$ of negative clauses and the set $\{C_1, \ldots, C_{n_d}\}$ of definite clauses. A sequence of graph expansion steps

$$\bot \underset{(N_1,\ldots,N_{n_n})}{\Rightarrow}{}^{\sigma_{id}} G_0 \overset{(\sigma_1^1,\ldots,\sigma_{k_1}^1)}{\underset{\oplus\ (C_1^1,\ldots,C_{k_1}^1)}{\Rightarrow}} G_1 \cdots G_{p-1} \overset{(\sigma_1^p,\ldots,\sigma_{k_p}^p)}{\underset{\oplus\ (C_1^p,\ldots,C_{k_p}^p)}{\Rightarrow}} G_p,$$

where $\bot$ stands for *nodefalse*, is called an *H-derivation*. If, in addition, G_p is an unsatisfiable H-graph, the derivation is called an *H-refutation*.

Any unsatisfiable H-graph, by definition, contains a pebbling from *nodetrue* to *nodefalse*. The basic idea of the argument is the following: from an H-refutation

$$\bot \underset{(N_1,\ldots,N_{n_n})}{\Rightarrow}{}^{\sigma_{id}} G_0 \overset{(\sigma_1^1,\ldots,\sigma_{k_1}^1)}{\underset{\oplus\ (C_1^1,\ldots,C_{k_1}^1)}{\Rightarrow}} G_1 \cdots G_{p-1} \overset{(\sigma_1^p,\ldots,\sigma_{k_p}^p)}{\underset{\oplus\ (C_1^p,\ldots,C_{k_p}^p)}{\Rightarrow}} G_p,$$

it is possible to extract a sequence of *simple* graph expansion steps

$$\bot \underset{N_b}{\Rightarrow}{}^{\sigma_{id}} G_0' \underset{C_{i_1}}{\Rightarrow}{}^{\sigma_{i_1}^1} G_1' \cdots G_{p-1}' \underset{C_{i_p}}{\Rightarrow}{}^{\sigma_{i_p}^p} G_p',$$

where $1 \le i_j \le k_j$, and $1 \le j \le p$, such that:

(1) $N_b \in \{N_1, \ldots, N_{n_n}\}$ is a distinguished negative clause, and G_0' is the corresponding graph.

(2) the node X_{j-1} chosen for expansion in each graph expansion step unifies with the head of only *one* definite clause (i.e., simple),

(3) G_j' is the subgraph $\sigma_{i_j}(G|age(X_{j-1}))[\sigma_{i_j}(X_{j-1}) \leftarrow \sigma_{i_j}(G_{C_{i_j}})]$ of G_j, where X_{j-1} is the node of G_{j-1} chosen for expansion, and

(4) the graph G_p' is unsatisfiable.

In words, for logic program P, an unsatisfiable H-graph G_p constructed by the HORNLOG proof procedure consists of many subgraphs, defined by age, and for any subgraph which yields a pebbling of *nodefalse*, this subgraph can be reconstructed by a series of simple graph expansion steps.

To address to question of a correct answer substitution, we proceed by defining the meaning of a logic program.

DEFINITION 5.2. The model-theoretic semantics for logic program P with query $Q = \exists z_1 \ldots \exists z_n Q'$, where $Q' = (\neg H_1 \vee \cdots \vee \neg H_m)$, each H_i a Horn clause, is defined as the set

$$D(P, Q) = \cup\Big\{\big\{(t_1^1, \ldots, t_n^1), \ldots, (t_1^k, \ldots, t_n^k)\big\},\ k \ge 1 \Big|$$
$$\models P \supset Q'[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q'[t_1^k/z_1, \ldots, t_n^k/z_n]\Big\},$$

where each $(t_1^i, \ldots, t_n^i)$, $1 \le i \le k$, is an n-tuple of terms (possibly containing variables) from the Herbrand universe for P and Q.

The standard Herbrand model consisting of a positive subset of the Herbrand base is not sufficiently rich to capture the semantics of disjunctions of substitutions instances of the query formula, as shown in the examples of Section 3.

The correctness of the procedure for computing answer substitutions can be argued as follows. Let P be a logic program consisting of a finite set of Horn clauses and with query $Q = \exists z_1 \ldots \exists z_n Q'$, where $Q' = (\neg H_1 \vee \cdots \vee \neg H_m)$, every H_j a Horn clause. Let the sequence of simple expansion steps

$$\bot \underset{N_b}{\Rightarrow}^{\sigma_{id}} G_0 \underset{C_1}{\Rightarrow}^{\sigma_1} G_1 \cdots G_{p-1} \underset{C_p}{\Rightarrow}^{\sigma_p} G_p$$

be extracted from an H-derivation for P and Q as described above. If $\langle H_{i_1}, \ldots, H_{i_k} \rangle$ is the subsequence of $\langle N_b, C_1, \ldots, C_p \rangle$ consisting of clauses in $\{H_1, \ldots, H_m\}$, then it can be shown that

$$\models P \supset \neg\theta_{i_1}(H_{i_1}) \vee \cdots \vee \neg\theta_{i_k}(H_{i_k}), \qquad (*)$$

where $\theta_j = \sigma_j \circ \theta_{j+1}$, for every j, $1 \leq j \leq p-1$, and $\theta_p = \sigma_p$.

Since each substitution θ_{i_j} in $P \supset \neg\theta_{i_1}(H_{i_1}) \vee \cdots \vee \theta_{i_k}(H_{i_k})$ corresponds to a tuple of terms $(t_1^j, \ldots, t_n^j)$, and since $Q' = (\neg H_1 \vee \cdots \vee \neg H_m)$, we have

$$\models \neg H_{i_1}[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee \neg H_{i_k}[t_1^k/z_1, \ldots, t_n^k/z_n] \supset Q'[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q'[t_1^k/z_1, \ldots, t_n^k/z_n],$$

and, together with $(*)$, this implies that

$$\models P \supset Q'[t_1^1/z_1, \ldots, t_n^1/z_n] \vee \cdots \vee Q'[t_1^k/z_1, \ldots, t_n^k/z_n],$$

where $\{(t_1^1, \ldots, t_n^1), \ldots, (t_1^k, \ldots, t_n^k)\}$ is the set of tuples of terms corresponding to the disjunctive answer. That is, every set of tuples of terms returned by an H-refutation belongs to $D(P, Q)$.

Note that the above tells us when disjunctive answers actually arise. Disjunctive answers arise when the query Q contains at least one disjunct $\neg H_j$ with a negative literal (equivalently, H_j is a definite clause), and when such clause(s) are used in at least two expansion steps in the sequence of graph expansions (that is, the sequence $\langle H_{i_1}, \ldots, H_{i_k} \rangle$ has at least two elements).

In order to show the completeness of the method, we proceed as follows. Recall that P is unsatisfiable, iff, by the Skolem-Herbrand-Gödel theorem, a set $\mathscr{P}$ of ground substitution instances of clauses in P is unsatisfiable, Now, $\mathscr{P}$ corresponds to a graph $\mathscr{G}$ whose nodes are the ground instances in $\mathscr{P}$, and $\mathscr{P}$ is unsatisfiable iff there is a pebbling of *nodefalse* from *nodetrue* in the graph $\mathscr{G}$. The trick then, is to show that $\mathscr{G}$ can be "lifted" to a graph G that can be obtained in a sequence of simple expansion steps, and such that $\mathscr{G}$ is the homomorphic image of G in this sequence.

The details of the proofs of soundness and completeness of the HORNLOG method can be found in [14, 27].

In previous attempts to proving the completeness of the HORNLOG method, we tried to use the fixed-point method of Apt and Van Emden [2], but without success. The main difficulty is that their approach does not seem to account easily for disjunctive answers. We are investigating whether a reasonable fixed-point semantics can be given for HORNLOG. In this regard, recent results of Fitting [10] may turn out to be helpful. In a recent paper [3], Apt, Blair, and Walker develop a theory of fixed

points of nonmonotonic operators, and show that it provides the declarative semantics of a certain class of programs where negative literals are allowed in the premise of a clause (the *stratified programs*). These results are very interesting, but it is not clear that they can be applied to HORNLOG programs, because negative assertions are not allowed.

6. COMPARISON WITH OTHER METHODS

The method presented in this paper has the flavor of a number of previously reported theorem proving methods that either use graphs in some manner or are based on the problem-reduction paradigm. However, we claim that there are significant differences between how other methods use various graph data structures and this problem-solving paradigm, and the method of HORNLOG.

The first and most obvious difference is that the other methods are full first-order theorem provers, whereas the HORNLOG method applies only to the Horn-clause logic subset. This is an important distinction, since the complexity of the satisfiability problem for arbitrary clauses is NP-complete even in the ground case, and the *traverse* procedure shows the unsatisfiability for ground Horn clauses in linear time.

The various connection-graph methods [1, 5, 18, 27] use graphs whose nodes are *clauses*, and whose edges labeled with substitutions connect unifiable literals of opposite sign. The nodes of H-graphs are *atomic formulae*, and the edges are pairs (C, σ), where C is a clause name and σ a substitution. In the connection-graph method, a given graph evolves to another graph in the following way: A link is selected and deleted, and then the resolvent is added and linked to the previous graph. Also, the resulting graph is reduced by removing clauses containing an unlinked literal and deleting other clauses and links which become redundant as a result of the removal. In HORNLOG, a graph is rewritten to another by expansion steps. A set of clauses is shown to be unsatisfiable in the connection-graph method when the empty graph is reached. In HORNLOG, unsatisfiability is discovered when the H-graph is checked by the procedure *traverse*.

The MESON procedure [23] is also similar in spirit to HORNLOG. Both are instances of the problem-reduction paradigm, and both use sophisticated marking schemes of their data structures. However, the data structure used in MESON is an AND/OR tree and *not* a graph, and its satisfiability is a consequence of propagating the truth value of leaves in this tree rather than the use of a method similar to the procedure *traverse*. In this paper when we have made the distinction between an AND/OR tree in which nodes are marked as "already seen" or "identical to previously seen" nodes, we had the MESON procedure in mind.

The idea of using clauses as *rewrite rules* as in context-free grammars is not new and has been exploited by Sickel [28] and Chang and Slagle [5]. However, these two methods are described for arbitrary clauses and are therefore more complicated. Sickel associates an attribute grammar with a set of clauses, and this attribute grammar is used to generate refutations. Chang and Slagle first build a connection graph from the set of clauses, and associate with this graph a set of context-free rules. These rules are then used to generate *plans* as guides for the search for a refutation. Although somewhat similar in spirit, our method is technically different and more direct because it applies to Horn clauses.

It should be noted that an unsatisfiable H-graph does produce a *mating* as defined in Andrews [1]. However, the major difference is in the method by which it is obtained and checked. As a matter of fact, a major difference with all other methods is in the procedure for checking unsatisfiability of the final graph. Since *traverse* operates in linear time in the size of the graph, the cost of calling *traverse* is much smaller than the cost of expanding graphs, and consequently we can afford to call *traverse* after every expansion step without significant time overhead.

7. CONCLUSIONS

We have presented a new method for logic-programming interpreters based on graph rewriting and on a linear-time algorithm for showing the unsatisfiability of ground Horn clauses. The system is presently implemented in PASCAL on a VAX 11/785, and uses a powerful LALR(1)-parser constructor incorporating systematic error recovery [17].

The method applies to a class of logic programs that we have called general Horn-clause programs. The examples given in Section 3 illustrate both some of the expressive power and limitations of general Horn-clause programs. It is possible to state some forms of negation without recourse to the negation-by-failure semantics, as shown by Examples 3.1 and 3.2. More "complicated' forms of negation, for instance an embedded negation as in $p(\neg q(X))$, or a negated literal in the body of a clause as in $A :- B_1, \ldots, \neg B_i, \ldots, B_p$, are not expressible, however.

It is possible to extract indefinite answers, in the sense that the answer is a disjunction of substitutions, as shown in Examples 3.2, 3.3, and 3.4. It seems clear that for certain applications indefinite answers are practically more desirable than no answer at all. For example, in a medical domain, it is certainly better for a system to conclude that a patient suffers from *either* condition A *or* from condition B than for the system to conclude that the patient suffers from *no* condition because it cannot return a definite answer. Indeed, a disjunctive answer gives a list of possible answers, and often suggests the addition of new assertions that would improve the definiteness of a program. On the other hand, we admit that the underlying mathematical semantics of HORNLOG may not be quite as simple as systems which return only definite answers (i.e., so far, we do not have a least-fixed-point semantics).

Finally, it is even possible in some cases, though not convenient, to express information which is not logically equivalent to any set of Horn clauses, by using what can only be called "programming tricks", as shown in Example 3.4. The use and need for such tricks also points out the inevitable fact that general Horn-clause programs are a proper subset of the full first-order logic, and that there will be problems that cannot be conveniently expressed in this subset. It is possible to generalize the HORNLOG procedure to handle clauses with negative literals in the premise, but his leads to a refutation procedure whose computational complexity is prohibitive (just in the ground case, graphs of exponential size may have to be constructed).

An interesting approach for dealing with a class of formulae more general than Horn clauses is presented by Miller in [25]. Miller gets around the problem of

having to deal with indefinite answers in an astute fashion. His solution is to abandon classical provability and instead to use more "constructive" notions of provability, such as intuitionistic provability. In this way, definite answers can be ensured for a class of programs consisting of formulae that may contain implications or disjunctions. Miller also gives a least-fixed-point semantics in terms of Kripke models, and discusses a theory of modules. He defines an interpreter at a fairly abstract level, and neither the complexity of his method nor an actual procedure using unification is presented.

In this paper we have presented the method with a "theorem proving" flavor, in the sense that it is complete and uses a breadth-first search for answer substitutions. It is equally possible to present the method as a procedure which uses depth-first search and backtracking, in a manner similar to PROLOG. Such a version has also been implemented in PASCAL. In this version, we represent the graph as a stack of records, each component of which consists of two parts:

(1) a section of the H-graph related to either the initial H-graph as described above, or the subgraph of an expansion step, and

(2) a substitution.

In the place of the construction of the graph resulting from considering all clauses that are unifiable with the selected mature node, instead, as in interpreters for PROLOG, only the first clause in lexical order is considered. The substitutions are not applied, so that when backtracking occurs, any state of the graph can be reconstructed from the information held in the stack. The instantiated graph is reconstructed as the stack is used by *traverse* to test for unsatisfiability.

We chose to present the breadth-first version of HORNLOG in detail, rather than the backtracking version, for two reasons. Presenting the breadth-first version enhances the clarity of presentation of eliminating the need to consider a HORNLOG counterpart of an SLD tree built by the SLD-resolution method, and the details of its traversal. Each node in such a counterpart is an H-graph, and the leaves are unsatisfiable H-graphs. Since the method itself uses a data structure which is related to the SLD tree, we felt the *simultaneous* explanation of two such data structures would unnecessarily complicate the presentation. There is, in addition, a more fundamental reason that motivated our choice of presentation.

While H-graphs *can* be represented in the form of stacks, this representation is not very natural and significantly complicates the procedure. The problem is that the H-graph data structure has the property that previous stages of the expansion cannot easily be recovered. Stated in another way, it is difficult to undo an expansion step during backtracking. Our experience suggests that the complications necessary to the procedure in order to make such a recovery possible cause it to compare unfavorably with SLD resolution in a single-processor environment.

In a parallel environment, however, we think the HORNLOG method will compare favorably with SLD resolution. A fundamental parallel interpretation seems immediate in both the graph-expansion step and the graph-traversal step. Since *any* nondead node can be chosen at *any* stage of the expansion cycle, conceptually, subject to the synchonization induced by a logic-programming language [7], for every expansion cycle on an architecture with n processors, n nodes could be

expanded simultaneously (one per processeor), and the n disjoint graphs merged. The resulting graph is tested for unsatisfiability by the *traverse* procedure, which, as was pointed out in [8], is itself modeled by the dataflow model. Since the H-graph data structure is a compact encoding for all possible ways of showing a formula unsatisfiable, a parallel strategy has a natural interpretation.

Note two differences between the HORNLOG method and SLD resolution in a parallel environment. First, it is possible for the construction of all branches of the SLD tree [2] in parallel to result in extensive duplication of information (the same substitution instance of a clause could occur in many different branches). The H-graph data structure represents duplicate information by merging nodes with duplicate labels. Second, nodes in an SLD tree are labeled with either negative *clauses* or the empty clause, while nodes in an H-graph are labeled with *atomic formulae*. It is possible (this is clearly a conjecture) that the finer "granularity" of the data structure used in HORNLOG coupled with the flexibility in the expansion process (each substitution instance of a predicate is attached to a processor, and any nondead node is available for expansion at any point in the derivation) could suggest a new approach to the problems induced by a logic-programming language on a parallel architecture. These are issues that we will have to explore before the conjecture that HORNLOG compares favorably with SLD resolution can be substantiated.

We also comment on the approach to equality used in HORNLOG. Based on work which extends SLD-resolution to admit equational Horn clauses [11, 15, 16], we have defined two extensions to the HORNLOG method presented here, based on unification modulo a set of equations, or E-unification. This first extension is general, in that it applies to arbitrary set of equational Horn clauses, but is not practical, as it assumes a procedure which gives an explicit sequence of substitutions for each E-unifier. The second extension, which we have called the $HE^{\dagger}$-refutation method, applies to any set of equational Horn clauses that admits a procedure enumerating a complete set of E-unifiers, and is complete for the set of logic programs containing clauses of the form $s \doteq t$, $Q :- P_1, \ldots, P_n$, or $:- P_1, \ldots, P_n$, where s and t are first-order terms, Q is a nonequational atomic formula, and $P_1, \ldots, P_n$ are either equational or nonequational atomic formulae. This is an important class of equational logic programs, in that it subsumes the paradigms of functional, logic, and equational programming.

It may also be possible to adapt our graph-based method to make it more incremental. By this, we mean that we would like to be able to reuse as much as possible a graph built by the procedure and found unsatisfiable, when the set of input clauses is changed, without having to go through previous expansion steps again. This would seem particularly advantageous when the procedure returns indefinite answers. Indeed, in many cases, an indefinite answer suggests assertions that could be added to the original set.

APPENDIX A

In this appendix, we give the proof of Theorem 2.3. Recall the statement of Theorem 2.3: Consider a first-order language without equality having at least one constant. For any (finite) set P of universally quantified Horn clauses, the following

properties hold:

(i) For any m ($m \geq 2$) sentences $A_i = \exists y_1^i \ldots \exists y_{p_i}^i B_i$, where each B_i is a conjunction of atomic formulae, if

$$\models P \supset A_1 \vee A_2 \vee \cdots \vee A_m,$$

then for some i, $1 \leq i \leq m$, we have

$$\models P \supset A_i.$$

(ii) For any sentence $\exists z_1 \ldots \exists z_p Q$, where Q is a conjunction of atomic formulae, if

$$\models P \supset \exists z_1 \ldots \exists z_p Q,$$

then there is a p-tuple of ground terms $(t_1, \ldots, t_p)$ such that

$$\models P \supset Q[t_1/z_1, \ldots, t_p/z_p].$$

PROOF. Theorem 2.3 can be proved using the fact that Horn sentences are preserved under direct products of models (Chang and Keisler [4]), or proof-theoretically as in Gallier [12]. We give a model-theoretic proof because it also applies to a slightly more general case. In this proof, the following notation is used. $\mathscr{V}$ denotes a countable set of variables, $\mathscr{M}$ denotes a first-order structure, M denotes the underlying domain of this structure, and an *assignment* is any function $s: \mathscr{V} \to M$. Given a term t, a formula A, a first-order structure $\mathscr{M}$, and an assignment $s: \mathscr{V} \to M$, the value $t_{\mathscr{M}}[s]$ and the truth value of $A_{\mathscr{M}}[s]$ for s are defined in the usual way (see Gallier [12]). Recall that if A is a *sentence* (that is, it has no free variables), then $A_{\mathscr{M}}[s]$ does not depend on s. Given a formula A, a structure $\mathscr{M}$, and an assignment s, we write $\mathscr{M} \models A[s]$ iff $A_{\mathscr{M}}[s] = \mathbf{true}$, $\mathscr{M} \models A$ iff $A_{\mathscr{M}}[s] = \mathbf{true}$ for every assignment s, and $\models A$ iff $\mathscr{M} \models A$ for every structure $\mathscr{M}$. We also write $\mathscr{M} \not\models A[s]$ iff $A_{\mathscr{M}}[s] = \mathbf{false}$ for some assignment s, and we write $\not\models A$ iff $\mathscr{M} \not\models A$ for some structure $\mathscr{M}$. Note that if A is a sentence, then $\mathscr{M} \not\models A$ iff $\mathscr{M} \models \neg A$.

Let I be a nonempty set which will be used as an index set, and let $(A_i)_{i \in I}$ be an I-indexed family of nonempty sets. The cartesian product, denoted by $\prod(A_i)_{i \in I}$, is the set of all I-indexed sequences $f: I \to \bigcup_{i \in I} A_i$ such that, for each $i \in I$, $f(i) \in A_i$. Such I-sequences will also be denoted as $\langle f(i) \mid i \in I \rangle$. For each $i \in I$, let $\mathscr{M}_i$ be a structure. We define the *direct product* $\mathscr{M}$ of the $(\mathscr{M}_i)_{i \in I}$ as the structure defined as follows:

The domain of $\mathscr{M}$ is the cartesian product $\prod(M_i)_{i \in I}$.

(1) Every constant symbol c is interpreted as the I-sequence $\langle c_{\mathscr{M}_i} \mid i \in I \rangle$.

(2) Every function symbol f of rank $n > 0$ is interpreted as the function such that, for any n I-sequences $G^1 = \langle g^1(i) \mid i \in I \rangle, \ldots, G^n = \langle g^n(i) \mid i \in I \rangle$,

$$f_{\mathscr{M}}(G^1, \ldots, G^n) = \langle f_{\mathscr{M}_i}(g^1(i), \ldots, g^n(i)) \mid i \in I \rangle.$$

(3) For every predicate symbol q of rank $n \geq 0$, q is interpreted as the predicate such that, for any n I-sequences $G^1 = \langle g^1(i) \mid i \in I \rangle, \ldots, G^n = \langle g^n(i) \mid i \in I \rangle$,

$$q_{\mathscr{M}}(G^1, \ldots, G^n) = \mathbf{true} \quad \text{iff} \quad q_{\mathscr{M}_i}(g^1(i), \ldots, g^n(i)) = \mathbf{true} \quad \text{for all } i \in I.$$

The direct product $\mathscr{M}$ is also denoted by $\prod(\mathscr{M}_i)_{i \in I}$. If $I = \{1, \ldots, m\}$, note

that every assignments $s: \mathscr{V} \to \prod(M_i)_{i \in I}$ corresponds to a unique m-tuple $(s_1, \ldots, s_m)$ of assignments $s_i: \mathscr{V} \to M_i$.

We now prove the theorem. First, we prove (i). Assume that $\models P \supset A_1 \vee A_2 \vee \cdots \vee A_m$, but $\not\models P \supset A_i$ for every i, $1 \leq i \leq m$. Since P is a set of sentences and $A_1, \ldots, A_m$ are sentences, there are structures $\mathscr{M}_i$, $1 \leq i \leq m$, such that

$$\mathscr{M}_i \not\models P \supset A_i.$$

Hence, for every i, $1 \leq i \leq m$,

$$\mathscr{M}_i \models P,$$

and

$$\mathscr{M}_i \models \neg A_i.$$

Since $A_i = \exists y_1^i \ldots \exists y_{p_i}^i B_i$, where B_i is a conjunction of atomic formulae, $\neg A_i$ is of the form $\neg A_i = \forall y_1^i \ldots \forall y_{p_i}^i (\neg B_1^i \vee \cdots \vee \neg B_{m_i}^i)$, where the B_j^i are atomic. Now, for every i, $1 \leq i \leq m$, $\mathscr{M}_i \models \neg A_i$ iff for every assignment $s_i: \mathscr{V} \to M_i$, $\mathscr{M}_i \models (\neg B_1^i \vee \cdots \vee \neg B_{m_i}^i)[s_i]$. Hence, for every i, $1 \leq i \leq m$, and for every s_i, there is some j_i, $1 \leq j_i \leq m_i$, such that

$$\mathscr{M}_i \models \neg B_{j_i}^i[s_i]. \qquad (*)$$

Note that for any atomic formula B and any assignment $(s_1, \ldots, s_m): \mathscr{V} \to \prod(M_i)_{i \in I}$, we have

$$\prod(\mathscr{M}_i)_{i \in I} \models \neg B[(s_1, \ldots, s_m)]$$

iff

$$\prod(\mathscr{M}_i)_{i \in I} \not\models B[(s_1, \ldots, s_m)]$$

iff (by the definition of validity in a direct product)

$$\mathscr{M}_i \not\models B[s_i] \qquad \text{for some } i, \quad 1 \leq i \leq m,$$

iff

$$\mathscr{M}_i \models \neg B[s_i] \qquad \text{for some } i, \quad 1 \leq i \leq m.$$

Applying the above observation to $\neg B_{j_i}^i[s_i]$, by $(*)$, we have

$$\prod(\mathscr{M}_i)_{i \in I} \models \neg B_{j_i}^i[(s_1, \ldots, s_m)],$$

that is,

$$\prod(\mathscr{M}_i)_{i \in I} \models \neg B_i[(s_1, \ldots, s_m)],$$

for every i, $1 \leq i \leq m$. Since the above argument holds for any arbitrary $s_1, \ldots, s_m$, we have shown that for every i, $1 \leq i \leq m$, we have

$$\prod(\mathscr{M}_i)_{i \in I} \models \neg A_i.$$

Since P is a set of Horn sentences, and sets of Horn sentences are preserved under direct products [4],

$$\mathscr{M}_i \models P \qquad \text{for every } i, \quad 1 \leq i \leq m$$

implies that

$$\prod(\mathcal{M}_i)_{i\in I} \models P.$$

But then, we have shown that

$$\prod(\mathcal{M}_i)_{i\in I} \models P \wedge \neg A_1 \wedge \neg A_2 \wedge \cdots \wedge \neg A_m,$$

contradicting the fact that

$$\models P \supset A_1 \vee A_2 \vee \cdots \vee A_m.$$

Hence, we must have $\models P \supset A_i$ for some i, $1 \leq i \leq m$.

We now prove (ii). Let HT be the Herbrand universe, that is, the set of all terms built up from constant and function symbols in the language. Assume that $\models P \supset \exists z_1 \ldots \exists z_p Q$, but that for every p-tuple of terms $(t_1, \ldots, t_p) \in HT^p$, we have $\not\models P \supset Q[t_1/z_1, \ldots, t_p/z_p]$. Then, for every p-tuple of ground terms $t = (t_1, \ldots, t_p) \in HT^p$, there is a structure $\mathcal{M}_t$ such that

$$\mathcal{M}_t \models P \wedge \neg Q[t_1/z_1, \ldots, t_p/z_p]. \qquad (**)$$

For simplicity of notation, for each $t = (t_1, \ldots, t_p) \in HT^p$, let us denote $Q[t_1/z_1, \ldots, t_p/z_p]$ as $Q(t)$. From $(**)$, for every $t \in HT^p$, we have

$$\mathcal{M}_t \models P \quad \text{and} \quad \mathcal{M}_t \models \neg Q(t).$$

Now, we use two facts:

(1) For any formula $B = B_1 \wedge \cdots \wedge B_m$, where $B_1, \ldots, B_m$ are atomic formulae, for any assignment $\langle s_i \mid i \in I \rangle : \mathcal{V} \to \prod(M_i)_{i\in I}$, we have

$$\prod(\mathcal{M}_i)_{i\in I} \models \neg B[\langle s_i \mid i \in I \rangle]$$

iff, for some j, $1 \leq j \leq m$,

$$\prod(\mathcal{M}_i)_{i\in I} \models \neg B_j[\langle s_i \mid i \in I \rangle]$$

iff, for some j, $1 \leq j \leq m$,

$$\prod(\mathcal{M}_i)_{i\in I} \not\models B_j[\langle s_i \mid i \in I \rangle]$$

iff, for some j, $1 \leq j \leq m$ (by the definition of validity in a direct product),

$$\mathcal{M}_i \not\models B_j[s_i] \quad \text{for some} \quad i \in I,$$

iff, for some j, $1 \leq j \leq m$,

$$\mathcal{M}_i \models \neg B_j[s_i] \quad \text{for some} \quad i \in I,$$

iff

$$\mathcal{M}_i \models \neg B[s_i] \quad \text{for some} \quad i \in I.$$

(2) Horn sentences are preserved under direct products.

Let $I = HT^p$. Since P is a set of Horn clauses, and since for every $t \in I$ we have

$$\mathcal{M}_t \models P,$$

by fact (2) we have

$$\prod(\mathcal{M}_i)_{i\in I} \models P.$$

Since $Q(t) = Q_1(t) \wedge \cdots \wedge Q_m(t)$ for some atomic formulae $Q_1, \ldots, Q_m$, and for every $t \in I$ we have

$$\mathcal{M}_t \vDash \neg Q(t),$$

by fact (1) we have

$$\prod(\mathcal{M}_i)_{i \in I} \vDash \neg Q(t) \qquad \text{for every} \quad t \in I.$$

Hence, we have shown that:

(3) The set $P \cup \{\neg Q(t) \mid t \in I\}$ is satisfied in $\prod(\mathcal{M}_i)_{i \in I}$.

On the other hand, since $\vDash P \supset \exists z_1 \ldots \exists z_p Q$, the set $P \cup \{\forall z_1 \ldots \forall z_p \neg Q\}$ is unsatisfiable. We claim that this implies that:

(4) The set $P \cup \{\neg Q(t) \mid t \in I\}$ is unsatisfiable.

Note that (4) contradicts (3), and this will establish part (ii) of the theorem. Since all sentences in $P \cup \{\forall z_1 \ldots \forall z_p \neg Q\}$ are clauses with no existential quantifiers, it is well known [12] that a sentence in negation normal form with no existential quantifiers is valid in some structure iff it is valid in some Herbrand structure, that is, a structure whose domain is HT. If $P \cup \{\neg Q(t) \mid t \in I\}$ were satisfiable, it would be satisfied in some Herbrand structure $\mathcal{M}$. But then, since the domain of $\mathcal{M}$ is the set HT of Herbrand terms, by the definition of validity of a universal formula, the fact that $\mathcal{M} \vDash A$ for every $A \in P \cup \{\neg Q(t) \mid t \in I\}$ implies that $\mathcal{M} \vDash P \wedge \forall z_1 \ldots \forall z_p \neg Q$, contradicting the unsatisfiability of $P \cup \{\forall z_1 \ldots \forall z_p \neg Q\}$. Hence, (4) holds.

Since the assumption that $\nvDash P \supset Q(t)$ for every $t \in HT^p$ leads to a contradiction, we must have $\vDash P \supset Q(t)$ for some $t \in HT^p$. □

Corollary. Let P be a conjunction of universal Horn clauses over a first-order language without equality having at least one constant. For any finite disjunction $A_1 \vee \cdots \vee A_m$ *of sentences of the form* $A_i = \exists y_1 \ldots \exists y_{p_i} B_i$, *where* B_i *is a conjunction of atomic formulae, if*

$$\vDash P \supset A_1 \vee \cdots \vee A_m,$$

then there is some i, $1 \leq i \leq m$, *and some tuple of ground terms* $(t_1, \ldots, t_{p_i})$, *such that*

$$\vDash P \supset B_i\left[t_1/z_1, \ldots, t_{p_i}/z_{p_i}\right].$$

PROOF. Immediate by Theorem 2.3. □

REMARK.

(1) A shorter proof of part (ii) of Theorem 2.3 can be given from part (i) of Theorem 2.3 and the Skolem-Herbrand-Gödel theorem. The proof that we have given uses more basic principles and shows the central role of the preservation under direct products. In effect, we have proven directly a special version of the Skolem-Herbrand-Gödel theorem for (universal) Horn formulae.

(2) The proof of part (i) applies to any set of sentences preserved under direct products. This includes sentences containing existential quantifiers, and

sentences not equivalent to any Horn sentences [4]. Part (ii) holds for any universal set of sentences preserved under direct products. However, by McKinsey's theorem [4], such a set of sentences has a set of axioms consisting of universal Horn sentences. Hence, it is likely that Theorem 2.3 only holds for sets of universal Horn sentences, and we conjecture that this is so.

Our proof technique also allows us to show that Theorem 2.3 holds for languages with equality. However, due to the lack of space, this generalization is presented in Ref. [14].

We would like to thank the referee for his careful review, and for making numerous suggestions which led to improvements in the presentation, particularly regarding the section on soundness and completeness. We would also like to thank Ken McAloon and Gopalan Nadathur for incisive comments and helpful suggestions given while this paper was being written.

REFERENCES

1. Andrews, Peter, Theorem Proving via General Matings, *J. Assoc. Comput. Mach.* 28(2):193–214 (1981).
2. Apt, K. R. and Van Emden, M. H., Contributions to the Theory of Logic Programming, *J. Assoc. Comput. Mach.* 29(3):841–862 (July 1982).
3. Apt, K. R., Blair, H., and Walker, A., Towards a Theory of Declarative Knowledge, Technical Report, IBM T. J. Watson Research Center, Yorktown Heights, N.Y. 10598, 1985.
4. Chang, C. C. and Keisler, J. H., *Model Theory*, North-Holland, 1978.
5. Chang, C. L. and Slagle, J. R., Using Rewriting Rules for Connection Graphs to Prove Theorems, *Artificial Intelligence* 12(2):159–178 (1979).
6. Clark, K. L., Negation as Failure, in: *Logic and Databases*, Plenum, 1978, pp. 293–322.
7. Connery, J. and Kibler, D., Parallel Interpretation of Logic Programs, in: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Vol. 1, pp. 163–170.
8. Dowling, W. P. and Gallier, J. H. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae, *J. Logic Programming* 1(3):267–284 (1984).
9. Fay, M., First-order Unification in an Equational Theory, in: *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Tex., 1979.
10. Fitting, M., A Kripke/Kleene Semantics for Logic Programs, *J. Logic Programming* 2(4):295–312 (1985).
11. Gallier, J. H., Fast Algorithms for Testing Unsatisfiability of Ground Horn Clauses with Equations. *J. Symb. Comput.*, submitted for publication.
12. Gallier, J. H., *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper and Row, 1986.
13. Gallier, J. H. and Raatz, S., Logic Programming and Graph Rewriting, in: *1985 IEEE Symposium on Logic Programming*, Boston, pp. 208–219.
14. Gallier, J. H. and Raatz, S., Theoretical Results for the HORNLOG Procedure, Technical Report, Univ. of Pennsylvania, Philadelphia, 1986.
15. Gallier, J. H. and Raatz, S., SLD-Resolution Methods for Horn Clauses with Equality Based on *E*-Unification, in: *1986 IEEE Symposium on Logic Programming*, Salt Lake City, Utah, to appear.
16. Gallier, J. H. and Raatz, S., Extending SLD-Resolution to Equational Horn Clauses using *E*-unification, *J. Logic Programming*, to appear.

17. Jalili, F. and Gallier, J. H., Building Friendly Parsers, in: *POPL 9*, Albuquerque, N. Mex. 1982, pp. 197–206.
18. Kowalski, R. A., A Proof Procedure Using Connection Graphs, *J. Assoc. Comput. Mach.* 22(4):572–595 (1975).
19. Kowalski, R. A., *Logic for Problem Solving*, Elsevier North-Holland, 1979.
20. Kowalski, R. A. and Van Emden, M. H., The Semantics of Predicate Logic as a Programming Language, *J. Assoc. Comput. Mach.* 23(4):733–742 (1976).
21. Kowalski, R. A. and Kuehner, D., Linear Resolution with Selection Function, *Artificial Intelligence* 2:227–260 (1970).
22. Lloyd, J. W., *Foundations of Logic Programming*, Springer, New York, 1984.
23. Loveland, D., *Automated Theorem Proving: A Logical Basis*, North-Holland, 1978.
24. Miller, D. and Nadathur, G., Higher-order Logic Programming, in: *Proceedings of Third International Conference on Logic Programming*, London, 1986.
25. Miller, D., A Theory of Modules for Logic Programming, in: *1986 IEEE Symposium on Logic Programming*, Salt Lake City, Utah, to appear.
26. Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Press, 1980.
27. Raatz, S., Aspects of a Graph-Based Proof Procedure for Horn Clauses, Ph. D. Dissertation, University of Pennsylvania, Dept. of Computer and Information Science, 1987.
28. Sickel, S., Formal Grammars as Models of Logic Derivations, in: *Proceedings of IJCAI-77*, 1977, pp. 544–551.