SATCHMOREBID: SATCHMO(RE) with BIDirectional relevancy

Article *in* New Generation Computing · December 2001

DOI: 10.1007/BF03037473 · Source: CiteSeer

2 authors, including:

Adnan Yahya
Birzeit University
41 PUBLICATIONS **439** CITATIONS

SATCHMOREBID: SATCHMO(RE) with BIDirectional Relevancy

Donald W. LOVELAND[1] and Adnan H. YAHYA[1,2]

[1] *Department of Computer Science, Duke University, Durham, NC 27708 USA*

[2] *Electrical Engineering Department, Birzeit University, Birzeit, Palestine*

{dwl,yahya}@cs.duke.edu

Received 15 September 2000

Abstract SATCHMORE was introduced as a mechanism to integrate relevancy testing with the model-generation theorem prover SATCHMO. This made it possible to avoid invoking some clauses that appear in no refutation, which was a major drawback of the SATCHMO approach. SATCHMORE relevancy, however, is driven by the entire set of negative clauses and no distinction is accorded to the query negation. Under unfavorable circumstances, such as in the presence of large amounts of negative data, this can reduce the efficiency of SATCHMORE. In this paper we introduce a further refinement of SATCHMO called SATCHMOREBID: SATCHMORE with BIDirectional relevancy. SATCHMOREBID uses only the negation of the query for relevancy determination at the start. Other negative clauses are introduced on demand and only if a refutation is not possible using the current set of negative clauses. The search for the relevant negative clauses is performed in a forward chaining mode as opposed to relevancy propagation in SATCHMORE which is based on backward chaining. SATCHMOREBID is shown to be refutationally sound and complete. Experiments on a prototype SATCHMOREBID implementation point to its potential to enhance the efficiency of the query answering process in disjunctive databases.

Keywords Disjunctive Deductive Databases , Query Answering, Bidirectional Search, Model Generation Theorem Proving, Relevancy.

§1 Introduction

The common approach to query answering in deductive databases is to find a refutation for the set of clauses representing the database and the negation of the query. In disjunctive databases, query processing generally has a high computational complexity [6]. To achieve acceptable performance several approaches were suggested. Some are bottom-up in the sense that they start from the facts of the database and derive new information until the negation of the query is contradicted. SATCHMO, introduced in [14] and formalized as a tableaux method in [2], is a representative of this class of algorithms. Other methods are based on top-down processing. They start from the query and generate subqueries until the generated queries are answerable by the database facts. The deduction tree [9], SLO [17] and the duality [21] methods represent this class. Others still combine top-down with bottom-up processing within a single system to improve performance. SATCHMORE [13] and Non-Horn Magic sets [7, 15], the extension of magic sets [1] to the disjunctive case, represent this approach. The debate on which is the best approach is still an open one [18, 20]. Bottom-up processing can explore a large search space by involving clauses that are irrelevant to the given query. Partial relevancy [19] and total relevancy [13] were introduced as means to restrict the processing to those clauses that are known to contribute to the refutation. Partial relevancy relies on having at least one atom of the head of a clause relevant before that clause can be used in a SATCHMO-style refutation while total relevancy requires all head atoms be relevant for this purpose. Both are refutationally sound and complete [19, 13], but total relevancy tends to explore a smaller search space. Therefore, we concentrate on algorithms employing total relevancy as represented by SATCHMORE[13].

Relevancy, as done by SATCHMORE and related systems [13, 7, 8, 15], starts with negative clauses. In keeping with database terminology, we hereafter refer to negative clauses (clauses of form $A \rightarrow \perp$) also as *integrity constraints (ICs)*, or simply *constraints*. The symbol $\perp$ denotes falsehood, so clauses of the form $(A \rightarrow \perp)$ encode the formula (*not* A). SATCHMORE uses all ICs and propagates the relevancy designation backwards from clause consequence (head) to antecedent (body) and to the matching head atoms of other clauses. That all ICs are used means that ICs with no relationship to Q still initiate relevance labeling[*1]. Strongly degraded performance can result from the presence of large numbers of unrelated ICs.

[*1] SATCHMO and SATCHMORE are presented as theorem proving programs and as such deal with refutations of sets of clauses. No advantage is sought from the special status of the query IC, but this is not central to a theorem prover.

To deal with the problem of propagating relevancy from all ICs we give a procedure SATCHMOREBID: SATCHMO(RE) with BIDirectional relevancy. Initially SATCHMOREBID uses the negation of the query and none of the database ICs as possible sources of relevancy. The latter are *promoted* to being sources of relevancy *on demand* by employing the non-negative clauses of the database to identify ICs that may contribute to advancing the refutation search. This is especially useful in database applications where the query negation is distinguished from the database ICs.

As in other approaches to relevancy testing, the benefit of the controlled introduction of ICs has to be weighed against the cost of finding the needed clauses. Experimental results will be given to show the potential advantages and limitations of the advanced method.

The rest of the paper is organized as follows. In the next section we give some needed definitions, recall the model-generation refutational procedure SATCHMO and the modification SATCHMORE. We discuss relevancy testing in SATCHMORE, its advantages and drawbacks. In Section 3 we introduce the concept of forward relevancy as a complement to traditional backward relevancy and combine them into an algorithm that performs bidirectional relevancy testing to allow for tighter refutation searches. We give an implementation of the suggested approach in the form of the SATCHMOREBID Prolog program and prove its refutational soundness and completeness. We present the results of our testing on several classes of examples to demonstrate the efficiency gains achievable. We comment on the advantages, limitations and the compromises involved in the SATCHMOREBID approach. In Section 4 we give our conclusions, compare our results with others reported in the literature, and mention some possible directions of future work.

§2 Preliminaries and Background Material

We assume familiarity with the basic concepts relating to disjunctive databases as e.g. in [11] as well as the theorem provers SATCHMO [14] and SATCHMORE [13] and limit ourselves to briefly recalling the basic material needed for the results presented here.

2.1 General

Definition 2.1
A *database*, S, is a set of clauses in implication form: $C = B_1 \wedge \ldots \wedge B_n \rightarrow A_1 \vee \cdots \vee A_m$, where $m, n \geq 0$ and the A_i and B_j are atoms in a First Order Language (FOL) $\mathcal{L}$ possibly with function symbols. C is *positive* if $n = 0$ (head is $\top$, *true*, empty) and *negative* or an *integrity constraint (IC)* if $m = 0$ (body is $\bot$, *false*, empty, `bottom`). By $Head(C)$ we denote the disjunction of atoms $A_1 \vee \cdots \vee A_m$ and by $Body(C)$ we denote the conjunction of atoms $B_1 \wedge \ldots \wedge B_n$. So $C = Body(C) \rightarrow Head(C)$.

The Herbrand base of S, HB_S, is the possibly infinite set of all ground atoms that can be formed using the predicate symbols and constants in $\mathcal{L}$. A *Herbrand interpretation*, I, is any subset of HB_S such that all ground atoms of I are assigned *true* and all other atoms of HB_S are assigned *false*. A clause is satisfied in an interpretation I if and only if at least one head atom is *true* in I or at least one body atom is *false* in I. A clause is *violated* in an interpretation I if and only if no head atom is *true* in I while all its body atoms are *true* in I. A Herbrand model of S, M, is a Herbrand interpretation such that $M \models S$ (all clauses of S are *true* in M). M is *minimal* if no proper subset of M is a model of S.

Definition 2.2
(Range-restriction) A clause C is *range-restricted (RR)* if every variable occurring in the head of C also appears in the body of C. A set of clauses is range-restricted if and only if every clause in the set is range-restricted.

Definition 2.3
(Query) A query is a conjunction of atoms. We assume that all queries are atomic (consist of a single atom Q). Any query of the form $C = B_1 \wedge \ldots \wedge B_n$ can be converted into the atomic query Q by adding the clause $B_1 \wedge \ldots \wedge B_n \rightarrow Q$ to the database.

The provability relation $S \vdash A$, where S is a clause set and A is a conjunction (disjunction) of atoms, will be Prolog provability throughout this paper.

2.2 SATCHMO

SATCHMO and its variations [14, 2] are model-generation based refutation logic theorem provers applicable to the class of range restricted clause sets. Starting from the empty interpretation SATCHMO operates by attempting to construct a model of its input clause set by splitting on (ground instances of) clause heads if the clause body is satisfied in the current interpretation and the head is not. The result is a new set of interpretations resulting from expanding the current interpretation by an atom of the expansion clause at a time. A query Q is a logical consequence of a set of clauses S if and only if $S \cup \{Q \rightarrow \bot\}$ has no model.

We utilize a limited bidirectional proof search organization introduced in [14] and adopted in [13]. The given disjunctive range-restricted clause set is divided into two sets. The first, BC, the Backward-Chaining component consists of a decidable Horn Clause set that includes the ICs of the database and those representing the negation of the query. The second, FC, the Forward-Chaining component, contains the remaining elements of the database including all disjunctive clauses.

The following is a simplified formal description of SATCHMO [13]:

Definition 2.4

(SATCHMO) Given a set T of RR clauses, SATCHMO operates as follows:

For each set I of ground atoms (initially empty):

1. If $BC \cup I \vdash \bot$ then $BC \cup FC \cup I$ is unsatisfiable.
2. If $BC \cup I \not\vdash \bot$ then select a clause $C \in FC$ such that $BC \cup I \vdash Body(C)|\sigma$, where σ is a grounding substitution, and $BC \cup I \not\vdash Head(C)|\sigma$ (C is *violated*). If no such clause exists then a minimal model of $BC \cup I$ is a model of $BC \cup FC$. The set T is satisfiable.
3. For each atom A_i in $Head(C)|\sigma$ of the violated clause C call the procedure recursively with $BC \cup I'$, where $I' = I \cup \{A_i\}$ (and the same FC).

   If $BC \cup FC \cup I'$ is unsatisfiable for every A_i then $BC \cup FC \cup I$ is unsatisfiable.

   T is shown to be unsatisfiable by completing the recursive call on the first selected violated clause.

The set I of atoms in the above definition is called a *partial interpretation*.

The following Prolog code gives the SATCHMO program [14] .

```
?- op(1200,xfx,'--->').
unsatisfiable:-
  bottom.
unsatisfiable:-
  not satisfiable.

satisfiable:-
  is_violated(C),!,
  satisfy(C),
  satisfiable.
satisfiable.
is_violated(A,C):-
  (A --->C),A, not C.
```

```
satisfy(C):-
  component(X,C),
  asserta(X),
  write('Asserting:'), write(X), nl,
  on_backtracking(retract(X)),
  not bottom.

component(X,(Y;Z)):-
  !, (X=Y; component(X,Z)).
component(X,X).

on_backtracking(X).
on_backtracking(X):-
  X, !, fail.
```

SATCHMO uses ICs to discard partial interpretations when the latter become inconsistent with the constraints. This may be a disadvantage when SATCHMO is used for query answering, where the negation of the query is added to the database and a refutation is sought. Except at the final stage, SATCHMO computations do not involve the query; they are not goal-directed. This can be costly in many cases.

2.3 SATCHMORE

The major shortcoming of SATCHMO is its eagerness to expand violated clauses even when they have no contribution to the refutation being sought. To avoid this, the idea of introducing relevancy was suggested [19, 13]. Under the restriction, only clauses shown to be relevant are allowed to participate in the SATCHMO-style computation. Partial relevancy [19] allows a clause to be used in the computation if one of its head atoms is relevant to the query while under total relevancy [13] all head atoms are required to be relevant. The latter is the more interesting case and the one which is incorporated into SATCHMORE.

Definition 2.5
(Relevant Atom) Given a Horn clause set BC and a conjunction of atoms G, such that $BC \nvdash G$, then each leftmost atom of a node in the failed SLD-tree [10] for $BC \cup \{\bot : \bot G\}$ is *extended relevant* to the derivation of G from BC. A ground instance A_g of an extended relevant atom A is *relevant* if $BC \nvdash A_g$.

The leftmost atoms of a node in a SLD-tree are the body atoms that have been *expanded*, i.e. for which program clauses have been sought to satisfy these body atoms. For any particular failed body atom A, either A or an atom below A has not found a matching program clause. A failed body atom, marked

by SATCHMORE and considered extended relevant, has as bound variables only those variables bound by inheritance from the clause head. We will refer to non-ground atoms as relevant when all ground instances are relevant.

Consider the following example:

Example 2.1
We give only the BC clause set component.

BC:
```
bottom  :- p(X,Y),q(Y).
p(X,b)  :- r(X,Y).
r(a,a).
```

The extended relevant atoms are: $p(X,Y), r(X,Y), q(b)$.
The relevant atoms are: $p(a,a), p(b,a), p(b,b)$,$r(a,b), r(b,b), r(b,a), q(b)$.

Definition 2.6
(Relevant Clause) Given a set of (extended) relevant atoms R and a clause C in FC then C is an *(extended) relevant clause* if and only if $Head(C) \subseteq R$. That is, iff all head atoms of C are (extended) relevant.

The following is a simplified formal description of SATCHMORE [13]:

Definition 2.7
(SATCHMORE) Given a set T of RR clauses, SATCHMORE operates as follows:

For each set I of ground atoms (initially empty):

1. If $BC \cup I \vdash \bot$ then $BC \cup FC \cup I$ is unsatisfiable.
   In attempting to derive $\bot$ mark any extended relevant literals encountered.
2. If $BC \cup I \nvdash \bot$ then select an extended relevant clause $C \in FC$. If no such clause is found then the minimal model of $BC \cup I$ can be extended into a model of $BC \cup FC$.
3. If C is not *violated* then return to step 2 for another extended relevant clause.
   While checking for violation of C mark any extended relevant literals encountered.
4. If C is violated: $BC \cup I \vdash Body(C)|\sigma$, where σ is a grounding substitution, and $BC \cup I \nvdash Head(C)|\sigma$ then the ground instance $C|\sigma$ is relevant. For each atom A_i in $Head(C)|\sigma$ call the procedure recursively with backward-chaining component $BC \cup I'$, where $I' = I \cup \{A_i\}$ (and

the same FC).
If $BC \cup FC \cup I'$ is unsatisfiable for every A_i then $BC \cup FC \cup I$ is unsatisfiable.
T is shown to be unsatisfiable by completing the recursive call on the first selected violated relevant clause.

The following Prolog code gives SATCHMORE program.

```
?- op(1200,xfx,'--->').

unsatisfiable:-
  bottom.
unsatisfiable:-
  not satisfiable.

satisfiable:-
  is_relevant(A,C),
  is_violated(A,C),!,
  satisfy(C),
  satisfiable.
satisfiable.

is_violated(A,C):-
  A, not C.

is_relevant(A,C):-
  retract(new_mark),
  (A--->C), each_marked(C).
is_relevant(A,C):-
  new_mark,
  is_relevant(A,C).

satisfy(C):-
  component(X,C),
  (retract(marked(_)), fail; true),
  asserta(X),
  write('Asserting:'), write(X), nl,
  on_backtracking(retract(X)),
  not bottom.

component(X,(Y;Z)):-
  !, (X=Y; component(X,Z)).
component(X,X).

on_backtracking(X).
on_backtracking(X):-
  X, !, fail.

each_marked((C1;CRest)):-
  !, marked(C1),
  each_marked(CRest).
each_marked(C):-
  marked(C).


mark_unique(X):-
  (marked(Y), is_instance(X,Y),!;
   subsume_and_mark(X)).

subsume_and_mark(X):-
  marked(Y), is_instance(Y,X), retract(marked(Y)), fail.
subsume_and_mark(X):-
  (new_mark, !; asserta(new_mark)),
  assertz(marked(X)).

is_instance(X,Y):- not X=Y , !, fail.
is_instance(X,Y):- var(Y),!.
is_instance(X,Y):- nonvar(X),
  functor(X,F,N), functor(Y,F,N), inst_args(X,Y,N).

inst_args(_,_,0):- !.
inst_args(X,Y,N):- arg(N,X,Ax), arg(N,Y,Ay),
  is_instance(Ax,Ay), N1 is N-1, inst_args(X,Y,N1).
```

The step where SATCHMORE differs from SATCHMO is that it checks for a

relevant violated clause rather than simply a violated clause as a candidate for expansion. This is done through the invocation of `is_relevant(A,C)`, which precedes `is_violated(A,C)` in `satisfiable`. Propagation of extended relevancy takes place when the relevant clause is tested for violation. The mechanism for determining extended relevant atoms, that is, the appropriate entries in the failure tree, involves the *marking clause*, which uses the `mark_unique` predicate. Each atom A in the head of a FC clause has a marking clause at the end of BC whose head subsumes A. A marking clause is called after failure to find any other eligible clause in BC. It executes a check for previously marked atoms so as to retain the most general instance, and, if appropriate, records the new atom as marked. Example 2.2 includes the appropriate marking clauses, which are omitted from further examples.

Due to the exponential nature of answer computations in disjunctive theories, the relevancy restriction often results in better performance despite the relevancy testing overhead. This is shown in detail in [13].

The following example taken from [13] demonstrates the possible gains resulting from relevancy testing employed by SATCHMORE.

Example 2.2

The following set of clauses is in the proper input format.

BC:
```
bottom :-p(c,X,Y).
bottom :-q(X,c,Z).
bottom :-r(X,Y,c).
t(X)     :- mark_unique(t(X)), fail.
p(X,Y,Z) :- mark_unique(p(X,Y,Z)), fail.
q(X,Y,Z) :- mark_unique(q(X,Y,Z)), fail.
r(X,Y,Z) :- mark_unique(r(X,Y,Z)), fail.
```

FC:
```
true ---> t(a).
true ---> t(b).
true ---> t(c).
t(X), t(Y), t(Z) ---> p(X,Y,Z); q(X,Y,Z); r(X,Y,Z).
```

A SATCHMO computation will examine all possible instances of the last clause with the resulting large number of cases considered. SATCHMORE on the other hand examines exactly the required instance, since only that instance is extended relevant and violated, to find a refutation. The difference in performance (resulting from expanding only the required instance of the last input

clause) is large [13].

2.4 Relevancy and Integrity Constraints (ICs)

The way SATCHMORE detects relevancy is by declaring all ICs, including the query negation, relevant and propagating this relevancy from clause heads to clause bodies. This is the result of propagating extended relevancy from $\bot$ at step 1 of the SATCHMORE algorithm. However, some of these ICs may have no contribution to the process of answering the query at hand and their presence may be undesirable as it drags members of FC into the computation that are not needed for finding a refutation.

Here is an example where a single irrelevant IC drags into "relevancy" useless components of the given set of clauses.

Example 2.3
Consider $S = S_r \cup S_{notr}$ where $S_r = \{C_1 = \top \rightarrow b, C_2 = b \rightarrow Q\}$, $S_{notr} = \{\mathbf{a_0} \rightarrow \bot, a_n \rightarrow a_{n-1}, ..., a_2 \rightarrow a_1, a_1 \rightarrow a_0, a_0 \vee a_1, a_1 \vee a_2, ...\}$ and the query Q. Then, a suitable input in the SATCHMORE style is:

$$BC = \{\bot : \bot Q, b, Q : \bot b, \bot : \bot a_0, a_{n-1} : \bot a_n, ..., a_1 : \bot a_2, a_0 : \bot a_1\}$$
$$FC = \{\top \rightarrow a_n; a_{n-1}, \top \rightarrow a_{n-1}; a_{n-2}, ..., \top \rightarrow a_2; a_1, \top \rightarrow a_1; a_0\}.$$

Including the IC of S_{notr} in BC will make all literals of S_{notr} relevant and expand the search space[*2] with all elements of FC.

However, clearly one cannot ignore the constraints in the refutation process; it becomes incomplete.

Example 2.4
Consider $S = \{C_1 = Q \vee b, C_2 = b \rightarrow c, C_3 = c \rightarrow \bot\}$ and the query Q.

Clearly none of the clauses of S is relevant if C_3 is ignored. If used then all the clauses become relevant and the refutation is achieved.

One way to avoid the problems illustrated here is to initially use the negation of the query and none of the other ICs. The latter are introduced *on demand* (and only after failing to find clauses relevant to the query alone). If we are unable to find a relevant and violated clause without introducing ICs, then we would like to invoke only the ICs that are necessary to advance the search for

[*2] The source of the elements of S_{notr} could be e.g. portions of databases independent of that to which the query is posed in a distributed environment.

a refutation by contributing to the relevancy of clauses needed in the refutation. We approximate this ideal in large part by seeking to extend selected partially relevant *seed* clauses to (totally) relevant clauses. This is done by determining sets of ICs that allow SATCHMORE to deduce that those seed clauses are relevant. This is done one seed clause at a time, with an unsatisfiability test done after each IC set is added.

The details are given in Section 3.

2.5 Sources of (Harmful) Negative Data

Several sources of negative data are possible that may drag into relevancy a large portion of the database. A partial list is:

1. Instances of nonground integrity constraints. Only certain (a small number of) instances of a constraint may be relevant to a given query. However, the irrelevant instances are still there and may render many of the database clauses relevant and they negatively affect the computation.

   Example 2.5
   Consider the following set of clauses.

   BC:
   ```
   bottom :-p(c,X,Y).
   bottom :-q(X,c,Z).
   bottom :-r(X,Y,c).
   ```

   FC:
   ```
   true ---> t(a).
   true ---> t(b).
   true ---> t(c).
   t(X), t(Y), t(Z) ---> p(X,Y,Z); q(X,Y,Z); r(X,Y,Z).

   v(X,Y,Z),s(X) ---> r(a,b,c).
   k(X,Y,Z) ---> v(X,Y,Z).
   m(X,Y,Z) ---> v(X,Y,Z).
   n(X,Y,Z) ---> v(X,Y,Z).
   j(X),j(Y),j(Z) ---> k(X,Y,Z);m(X,Y,Z);n(X,Y,Z).
   true ---> j(g).
   true ---> j(h).
   true ---> j(i).
   ```

   This is an expansion of Example 2.2. The added clauses are dragged

into the computation by having the irrelevant instance `r(a,b,c)` of the IC `bottom :-r(X,Y,c).` in the head of the first added clause `v(X,Y,Z),s(X) ---> r(a,b,c)`. In a sense this instance creates a relevancy link between the original and new components. Operating on the resulting set of clauses, SATCHMORE takes days to answer the query though it could do that in a fraction of a second without the added irrelevant clauses.

2. Combining knowledge bases: When the constraints in one base, say S_i, while useful for answering queries against S_i maybe (and generally are) of no use to queries against S_j, $j \neq i$. However, they may have the effect of making much of the other bases relevant and therefore participate in the computation.

Example 2.6

We present a class of clause sets, with a fixed BC and an indexed FC_i. For FC_i there is a fixed portion S_b (the first eight clauses) and an indexed portion we label NH_i for future reference.

BC:

```
bottom :- Q.
```

FC_i:

```
a,d---> e;f.                 true --->c.
c,e---> b.                   true --->Q;a.
d,f---> g.                   f,g--->bottom.
c--->d.                      b,e--->bottom.

ti,ri,mi---> si.             ti,ki,si--->bottom.
ti,ki---> mi;si.             ti,ki,mi--->bottom.
ji,ni---> si;ti.             ni,hi,ri--->bottom.
hi,mi---> si;ti.             ji,ki,ti--->bottom.
ri--->mi;ni.                 ji,ri,si--->bottom.
true --->ti;ri.              hi,ti,ji--->bottom.
true --->ki;pi.              ti,ri--->bottom.
true --->hi;ti;ji.           si,ni,mi--->bottom.
```

S_b is the base set of clauses in which the query Q is answerable and NH_i is the foreign NonHorn set of clauses which has nothing to do with the query.

SATCHMORE will activate both sets in its search for a refutation and the resulting timing of proving the query Q deteriorates rapidly as more

foreign bases are added. We will give our experimentation results at a later stage.

§3 SATCHMOREBID

In SATCHMORE, all ICs, independent of whether they originate in the database itself or query negation, participate in the computation on equal footing and relevancy is propagated from any of them. In defining our new procedure, SATCHMOREBID, we will use only a subset of the ICs; usually the initial IC set comes only from the query. The set of useful ICs is grown as the computation proceeds, and often a sufficient set of ICs is obtained before all ICs are invoked. To realize this we form two overlapping subsets BC_0 and FC from the given database S plus the query negation. FC is static and includes all elements of S (but not the query negation). BC_0 consists of the negation of the query in addition to any user-selected Horn clauses of S such that proof termination is assured. Usually no other ICs are included in BC_0, but the user may choose to include any IC clauses known to participate in the query deduction. At any given stage and for the current value of i, only ICs in BC_i and no other IC of FC are used in defining relevancy. Relevant atoms and clauses are always defined relative to the current BC_i, even when this is not explicitly mentioned. Non-negative clauses of FC are used in a forward chaining mode to detect ICs in FC that are later *promoted* to become elements of the next BC_i. ICs promoted to the backward chaining component are not removed from FC and therefore FC stays constant.

The ICs of BC_i are distinguished from the ICs of FC by their form. The forms are $\bot : \bot Body(C)$ and $Body(C) \rightarrow \bot$ respectively.
An IC $Body(C) \rightarrow \bot \in FC$ is *promoted* to be an element of BC_{i+1} by asserting the clause $\bot : \bot Body(C)$.

Definition 3.1
(Partially Relevant Clause; Atom) Given a set of (extended) relevant atoms R and a clause C in FC, then C is a *partially relevant clause* if and only if $Head(C) \cap R \neq \emptyset$ but C is not an (extended) relevant clause. A head atom A of a partially relevant clause is a *partially relevant atom* if and only if $A \notin R$.

Definition 3.2
(SATCHMOREBID) Given a RR database S, an atomic query Q and the set BC_0 defined as a Horn subset of S union $\{\bot : \bot Q\}$, SATCHMOREBID operates

as follows:

For each set I of ground atoms (initially empty) and the current BC_i (initially $i = 0$):

1. If $BC_i \cup I \vdash \perp$ then $BC_i \cup FC \cup I$ is unsatisfiable.
   In attempting to derive $\perp$ mark any extended relevant literals encountered.
2. If $BC_i \cup I \not\vdash \perp$ then select an *extended relevant* clause $C \in FC$. If no such clause is found then let $BC_{i+1} = BC_i \cup FR(BC_i, I)$, where $FR(BC_i, I)$ is the set of ICs in FC that are forward reachable[*3] from I and the set of partially relevant atoms relative to BC_i. If $BC_{i+1} = BC_i$ then $S \cup BC_0$ is satisfiable and the minimal model of $BC_i \cup I$ can be extended into a model of $S \cup BC_0$.
   Otherwise, $(BC_{i+1} \neq BC_i)$. Let $i = i + 1$; and go to Step 1.
3. If C is not violated then return to step 2 for another extended relevant clause.
   While checking for violation of C mark any extended relevant literals encountered.
4. If C is violated, where σ is a grounding substitution, and $BC_i \cup I \not\vdash Head(C)|\sigma$ then the ground instance $C|\sigma$ is relevant. For each atom A_i in $Head(C)|\sigma$ call the procedure recursively with backward-chaining component $BC_i \cup I'$, where $I' = I \cup \{A_i\}$ (and the same FC).
   If $BC_i \cup FC \cup I'$ is unsatisfiable for every A_i then $BC_i \cup FC \cup I$ is unsatisfiable.
   $S \cup BC_0$ is shown to be unsatisfiable by completing the recursive call on the first selected violated clause.

We comment on item 2 of the definition. On failing to find a refutation using SATCHMORE with the current BC_i SATCHMOREBID doesn't fail but attempts to expand BC_i into BC_{i+1} by promoting ICs from FC. This is done through computing $FR(BC_i, I)$, the set of ICs in FC that are forward reachable from I and the set of partially relevant atoms relative to BC_i. If this set is empty then the procedure fails and reports satisfiability. Otherwise, BC_i changes and the SATCHMORE code is invoked with BC_{i+1}. So only if both SATCHMORE with the current BC_i and the forward reachability search fails does the procedure report failure (satisfiability). Otherwise it makes another attempt at finding a refutation. Therefore, for a given BC_i and I it is well-defined to talk about the

[*3] This term is formally defined later.

SATCHMORE invocations in SATCHMOREBID, as we do frequently later in this paper.

3.1 SATCHMOREBID Program

We now list the SATCHMOREBID program. The input is in two (generally overlapping) components, BC and FC, where FC contains all clauses except the IC associated with the query. BC is a decidable subset of the Horn clauses, contains the query IC but generally does not include all the needed ICs. If more than one IC is included in BC then the query IC should appear first. The formats for the clauses of BC and FC are as previously stated, and as for SATCHMO and SATCHMORE.

The operation of SATCHMOREBID is as for SATCHMORE until no extended relevant clause can be found. Recall that an extended relevant clause has each head atom extended relevant, i.e. satisfying the `marked` predicate. The only variations from the original SATCHMORE code (other than the call to `part_relevant`) are the goal `clause(bottom, Q)` by which the query atom name is acquired and the assertion and possible retraction of `new_in_I(X)` for each new member `X` added to the partial interpretation I. These are of no use until `part_relevant` is invoked. (By *invoking* `part_relevant` we will mean the activation of the second clause of `part_relevant` due to a failing back to `part_relevant`, often called a REDO call to `part_relevant`. A call to `part_relevant` is a passthrough (NO OP) that allows SATCHMORE to proceed until completion or failure to find an extended relevant clause, which causes a REDO call to `part_relevant`.)

The purpose of `part_relevant` is to add to the set of ICs entered in BC to increase the chance that SATCHMORE will succeed. We seek to enrich the IC set in BC by chaining forward to the ICs needed. The forward chaining begins with a "seed", either a partially relevant clause or an element of I. Each head atom of a selected clause is matched with a body atom (the forward linking) of a clause of FC and all head atoms of the new clause again linked to other body atoms. (The query atom is excepted; see below.) This recursion forms a "fan" of paths from the seed to a collection of ICs of FC plus the query negation, some of which may be in BC already. Those that are not are added to BC. The fan of paths must all terminate in ICs for `part_relevant` to succeed, otherwise backtracking occurs, perhaps back so far as to select a new seed. That all paths terminate in ICs is a necessary condition for the seed clause to be determined

```
?- op(1200,xfx,'--->').

unsatisfiable:-
  bottom.
unsatisfiable:-
  clause(bottom,Q),
  not satisfiable.

satisfiable:-
  part_relevant,
  is_relevant(A,C),
  is_violated(A,C),!,
  satisfy(C),
  satisfiable.
satisfiable.

is_violated(A,C):-
  A, not C.

is_relevant(A,C):-
  retract(new_mark),
  (A--->C), each_marked(C).
is_relevant(A,C):-
  new_mark,
  is_relevant(A,C).

satisfy(C):-
  component(X,C),
  (retract(marked(_)), fail; true),
  asserta(X), asserta(new_in_I(X)),
  write('Asserting:'), write(X), nl,
  on_backtracking(X),
  not bottom.

component(X,(Y;Z)):-
  !, (X=Y; component(X,Z)).
component(X,X).

on_backtracking(X).
on_backtracking(X):-
  retract(X),
  (retract(new_in_I(X)); true),
  !, fail.

each_marked((C1;CRest)):-
  !, marked(C1),
  each_marked(CRest).
each_marked(C):- marked(C).

part_relevant.
part_relevant:-
  from_the_top,
  ((A--->C), one_marked(C); true_in_I(A,C)),
  forward(A,C),
  retract(new_IC),
  find_relevant_atoms.

true_in_I(true, element_of_I):-
  retract(new_in_I(element_of_I)).

one_marked(C):-
  component(X,C),
  marked(X), !.

forward(A,bottom):-
  !, (clause(bottom, A), !
     ;
     assertz((bottom :- A)),
        (new_IC; assertz(new_IC)),
        (added_IC; assertz(added_IC))).
forward(_,C):-
  each_forw(C).

each_forw((C1;CRest)):-
  !,(C1=Q; find_body(C1)),
  each_forw(CRest).
each_forw(C):-
 (C=Q; find_body(C)).

find_body(Ccomp):-
  (A--->C), ck_in_A(Ccomp,A),
   forward(A,C).

ck_in_A(Ccomp,A):-
  body_component(X,A),
  Ccomp=X.

body_component(X,(Y,Z)):-
  !, (X=Y; body_component(X,Z)).
body_component(X,X).

from_the_top.
from_the_top:- retract(added_IC), from_the_top.

find_relevant_atoms:- bottom, !.
find_relevant_atoms.
```

```
mark_unique(X):-
  (marked(Y), is_instance(X,Y), !;
   subsume_and_mark(X)).

subsume_and_mark(X):-
  marked(Y), is_instance(Y,X), retract(marked(Y)), fail.
subsume_and_mark(X):-
  (new_mark, !; asserta(new_mark)),
  assertz(marked(X)).

is_instance(X,Y):- not X=Y , !, fail.
is_instance(X,Y):- var(Y),!.
is_instance(X,Y):- nonvar(X),
  functor(X,F,N), functor(Y,F,N), inst_args(X,Y,N).

inst_args(_,_,0):- !.
inst_args(X,Y,N):- arg(N,X,Ax), arg(N,Y,Ay),
  is_instance(Ax,Ay), N1 is N-1, inst_args(X,Y,N1).
```

relevant by SATCHMORE, and constitutes an atomic step, in some sense, in the progress of SATCHMORE towards a refutation.

We now outline a walkthrough of an invocation of the procedure `part_relevant`. The procedure `from_the_top` is a passthrough when called; its purpose is to force a repeat of `part_relevant`. This may be needed if an IC has been added in a previous execution of `part_relevant` since new ICs can define new partially relevant clauses which in turn are seeds for discovery of new ICs. Recall that `part_relevant` does not succeed due to the discovery of one new IC, but rather must complete a fan of paths. Failing completion, backtracking occurs and new seeds may have to be processed.

The compound goal following `from_the_top` creates the seeds for the forward chaining. A seed may be a partially relevant clause, detected by the presence of a `marked` atom, or an element of I. If every clause in FC either fails `one_marked` or cannot identify an IC for each forward path, then elements of I are selected as seeds, structured as (`true ---> X`), for $X \in I$.

The next goal in `part_relevant` is `forward(A,C)`, which contains the basic recursion of the forward chaining. The first clause of the procedure `forward` handles the IC, checking for redundancy and then asserting the IC. The flags are then set; `added_IC` is used by `from_the_top`, discussed above. The flag `new_IC` indicates that at least one new IC has been defined, a requirement for success of `part_relevant`. Note that the purpose of `new_IC` is quite different from the flag `added_IC`; the former is turned on at most once per run of `part_relevant`, the latter is turned off with each REDO of `from_the_top`.

If the clause passed to `forward` is not an IC, then the second clause of

`forward` sends the clause head to `each_forw`, which in turn sends each head atom excepting the query atom to `find_body`. The query atom is excluded because its IC is not in FC, the only clause not in FC. No search for ICs to establish the relevance of the query is needed, of course. A more complex program can remove other atoms of `each_forw` so they do not reach `find_body`. Head atoms of paths originating in partially relevant clauses can be blocked if they are extended relevant. Relevant atoms already have paths to ICs, by the definition of relevance. Provable head atoms prevent the clause from being violated, so the clause cannot be on the path that uses the IC to establish relevancy for the seed head atom. However, no similar argument holds for elements of I, which use the same forward chaining mechanism. Failure to use the more complex program design may cost extra computation time, but we suspect that few unneeded ICs enter for this reason. This is partly because provable atoms either use elements of I (hence handled elsewhere) or will not connect to an IC as FC is consistent.

The procedure `find_body` finds a clause instance in FC with a body atom that agrees with the given atom. The discovered clause instance is in turn processed by `forward`, which perpetuates the recursion.

The recursion terminates with the discovery of an IC or with failure in `find_body` to find a clause to continue the forward chaining. The latter outcome triggers a backtracking where the main choice points are the sweep of FC in `find_body` and `part_relevant`. Exhaustion at these points results in the failure of `part_relevant`. Failure of `part_relevant` generally means that the clause set is satisfiable, but ICs could have been introduced that make BC unsatisfiable. This only arises if the database itself is inconsistent and catching this inconsistency would be a happenstance that all pertinent clauses were in BC at this point. No action is taken to detect this anomaly, although the addition of body `not bottom` to the second clause of `satisfiable` will detect the unsatisfiability of BC. Again, failure of `part_relevant` means satisfiability of $BC_0 \cup FC$ (this strange case excepted).

If the recursion does terminate with an IC, then `find_body` succeeds and `each_forw` may call another instance of `find_body`. Barring failure of `find_body`, discussed above, the call to `forward` in `part_relevant` eventually succeeds. Then `new_IC` is probed (and retracted). If a new IC was entered into BC during this invocation of `part_relevant` then a new attempt to prove `bottom` identifies any new (extended) relevant atoms defined by the new ICs in BC. Now `part_relevant` succeeds and control is returned to the SATCHMORE code for a new try at proving unsatisfiability.

3.2 Correctness of SATCHMOREBID

Until otherwise stated, all clause sets of this section are ground. Throughout the proofs of this section it is very important to be clear about relevancy and extended relevancy. We will use the term "extended relevancy" whenever that term is intended.

Lemma 3.1
(**SATCHMORE property**) When SATCHMORE fails with BC_i, then every (totally) relevant clause in FC (relative to BC_i) also has a body atom that is relevant.

Proof If some relevant clause has no relevant body atom, then all body atoms are provable, whereupon the clause is violated and a head atom entered into the partial interpretation. Such a head atom is not relevant, contradicting our assumption that the clause is relevant. ■

Definition 3.3
(**Forward Reached**) An atom is a *Forward Reached* (FR) atom if and only if it appears as an argument of the `find_body` procedure and is not relevant.

Starting with the head atoms of seed clauses, all head atoms that are passed to `forward`, except for `bottom`, are also passed to `find_body`. (We mentioned in the last subsection that a more complex program can filter out marked atoms when the seed of this forward chaining is a partially relevant clause.) The set of FR atoms includes all atoms of the current partial interpretation I as these atoms are seeds retrieved by `true_in_I`. These atoms cannot be relevant as their assertion makes them provable, but they might be extended relevant by happenstance (by an occurrence in a failed antecedent). Note that `bottom` is not a FR atom.

Lemma 3.2
(**Clauses with FR body atoms**) Given a set of RR clauses S and for a given BC_i and partial interpretation I, if both SATCHMORE and `part_relevant` fail then any clause in FC with a FR body atom has either

1. A FR head atom, or
2. A relevant body atom.

Proof By the construction of procedures `forward`, `each_forward` and `find_body`, every non-relevant head atom (excepting `bottom`) of a clause containing a FR body atom is a FR atom. Every clause with the FR body atom is processed

because the failure of `part_relevant` means that the first goal of `find_body` attempts a unification with all clauses of FC before it fails. If all the head atoms are relevant atoms then, by the SATCHMORE property, the clause also has a relevant body atom. Thus, the lemma holds for non-IC clauses. For an IC-clause that is in BC_i we note that the body always contains a relevant atom, so then condition 2 is satisfied. If an IC-clause is not in BC_i but contains an FR body atom then it must have been promoted to be a member of BC_{i+1}. But then $BC_{i+1} \neq BC_i$ and `part_relevant` would not have failed. A contradiction. Thus there is no IC-clause not in BC_i with an FR body atom. So, the lemma holds for all clauses in FC with a FR body atom. ■

Lemma 3.3
(`part_relevant` **success**) If FC is satisfiable, $FC \cup BC_0$ is unsatisfiable, SATCHMOREBID has defined BC_i, for some $i \geq 0$, in the processing of $FC \cup BC_0$, and `part_relevant` is reentered after a failure of SATCHMORE, then `part_relevant` will succeed with $BC_{i+1} \neq BC_i$.

Proof Because `part_relevant` cannot succeed without the addition of some IC-clause to BC_i, by explicit test, we need only show that `part_relevant` succeeds.

We suppose that `part_relevant` is reentered but does not succeed. From that we will show that then $FC \cup BC_0$ has a model, contradicting the unsatisfiability of $FC \cup BC_0$.

We define the interpretation M for $FC \cup BC_0$ by the following steps:

1. All relevant atoms are defined to be *false*. The relevant atoms are determined by the last failed run of SATCHMORE.
2. All FR atoms are defined to be *true*.
3. The minimum model M^* of FC provides the truth assignments to the remaining atoms.

Since FC is satisfiable one or more models exist for FC without the constraints of (1) and (2).

We now show that M is a model for $FC \cup BC_0$ under the assumption that $BC_{i+1} = BC_i$.

1. BC_0 clauses. This contains IC-clause `bottom:- Q`. The atom Q is relevant, hence *false* in M. The clause thus is *true* in M. All other clauses appear also in FC and are considered there.

2. For FC clauses, we need only be concerned with clauses that involve atoms with truth values changed from M^* as M^* is a model of FC.

   a. C is a clause of the form $Body(C) \rightarrow Head(C)$ and $Head(C)$ has at least one reassigned head atom. We need only be concerned with head atoms reassigned to *false* in M. But only the relevant atoms could have been reassigned to *false*. Thus we can assume that at least one head atom is relevant. If all head atoms are relevant then by the SATCHMORE property there is also a relevant body atom, and that is *false* in M. Such a clause is therefore *true* in M. If not all head atoms are relevant atoms then the clause is partially relevant, and by the code of `part_relevant` the non-relevant atoms are FR atoms. Such atoms are reassigned *true* in M, so such clauses are *true* in M.

   b. C is a clause of form $Body(C) \rightarrow Head(C)$ and $Body(C)$ has at least one reassigned body atom. We need only be concerned with the reassignment of a truth value to *true* in the body. Suppose that A has been reassigned to *true* in M. Then A must be a FR atom. By the Lemma 3.2, C then has either a FR head atom, with truth value *true* in M, or a relevant body atom, which has truth value *false* in M. In either case, C is *true* in M.

Having accounted for all cases we see that M is a model of $FC \cup BC_0$.

■

SATCHMO, SATCHMORE and SATCHMOREBID all have termination problems, but those of SATCHMOREBID differ from the others. SATCHMOREBID has termination problems with the forward relevancy checking of `part_relevant` and this can occur at the ground level. Clauses $a \perp \perp \perp > b$ and $b \perp \perp \perp > a$ together in FC can cause looping as can any more complex clause sets that incur looping. For databases, the focus of applications of SATCHMOREBID in our view, cycles are less likely than in theorem proving applications, but we must note the hazard. The protection against non-termination is that employed by SATCHMO [14] (at the first-order level), the use of level-saturation. This is in effect a breadth-first search (but most likely would be implemented as an iterative-deepening search) which is definitely more expensive than the depth-first style search employed by Prolog. Non-termination comes to all the above systems for first-order clause sets, so we do not view the added exposure to non-termination in `part_relevant` as a serious problem. Completeness re-

sults must assume the level-saturation version, although such versions will not be used in practice for most problems. Of course, it is necessary to make BC always terminating ("decidable") but this is easily done by careful selection of Horn clauses for BC.

Theorem 3.1

(**Ground completeness of SATCHMOREBID**) Let Q be an atomic query and S be a consistent range-restricted database. If $T = S \cup \{\bot : \bot Q\}$ is unsatisfiable and either T contains no cycles or SATCHMOREBID employs level-saturation to insure termination, then SATCHMOREBID succeeds (reports unsatisfiability).

Proof

The clause set T is represented by $BC_0 \cup FC$, where BC_0 contains clause $\bot : \bot Q$ and possibly other Horn clauses (including other ICs on occasion). (We assume BC_0 is decidable.) By Lemma 3.3 whenever both SATCHMORE and `part_relevant` fail, then BC_{i+1} properly contains BC_i. Failure of SATCHMOREBID requires a succession of failures of both SATCHMORE and `part_relevant`, for each failure that creates a new BC_{i+1} forces SATCHMORE to process the new representation $BC_{i+1} \cup FC$. ICs are thus added to the variants of BC until SATCHMORE succeeds, because the completeness of SATCHMORE assures detection of unsatisfiability when the full set of ICs of the unsatisfiable set $BC_0 \cup FC$ is active. ■

Note that in order for SATCHMOREBID to work correctly, we insist that the original set of clauses (without the query negation) be consistent. A test of this fact needs to be performed outside of SATCHMOREBID. In a sense, SATCHMOREBID is attempting to find a subset of the database that is inconsistent with the query negation. If such a component exists then the procedure reports success. Otherwise it reports failure (satisfiability), even if the database itself was inconsistent. While the latter is not logically correct it may be a feature that is useful for answering queries in the presence of inconsistent data. However, we don't pursue the details of this topic here.

Example 3.1

$T = \{\top \rightarrow Q_1 \vee c, \top \rightarrow Q_2 \vee f, c \rightarrow d \vee e, d \rightarrow \bot, e \rightarrow \bot, \top \rightarrow b, b \rightarrow c\}$.

The set is inconsistent. However, given the queries Q_1 and Q_2, using SATCHMOREBID one is able to find a refutation that involves Q_1 but none that involves Q_2. Note that SATCHMORE detects the inconsistency of the clause

set for both queries, and the clauses used exclude the queries.

We now consider the general first-order case. That is, the clause sets (or databases and queries) are derived from sets of formulas of first-order logic.

Theorem 3.2
(**Soundness of SATCHMOREBID**) If SATCHMOREBID succeeds using a Prolog with full unification on a first-order clause set $T = S \cup \{\bot : \bot Q\}$, where S is consistent then Q follows from S.

Proof SATCHMOREBID uses SATCHMORE code to process clause sets all of which are subsets of $BC_0 \cup FC$. Therefore, the soundness of SATCHMOREBID follows form the soundness of SATCHMORE. ■

We have already noted that care is needed to assure termination of SATCHMOREBID for arbitrary clause sets. First-order termination is a problem for all SATCHMO-based systems. See the discussion in [14].

Theorem 3.3
(**Completeness of SATCHMOREBID**) Given a level-saturated version of SATCHMOREBID using a Prolog with full unification, if a database S is a consistent set of first-order clauses and atomic query Q logically follows from S then SATCHMOREBID succeeds given the clause set $T = S \cup \{\bot : \bot Q\}$.

Proof Using a now-standard lifting argument the ground completeness theorem is used to yield the first-order theorem. See a standard text on resolution theory, such as [4] for details. ■

From preceding considerations in this paper, it is clear that the above theorem can be generalized to arbitrary unsatisfiable clause sets. We have stated the theorem to address the most common envisioned use of SATCHMOREBID.

3.3 Efficiency Considerations and Design Tradeoffs

The goal of SATCHMOREBID is to limit the number of ICs that participate in SATCHMORE-style computations on the assumption that the less the number of such clauses the more the computation is focused on the truly relevant clauses. The ideal situation is that only ICs needed for the refutation are promoted to BC_i. In an implementation the cost of filtering out undesired ICs should be weighed against the possible gains. The following are some of the points where one has to consider these tradeoffs and the choices adopted in the

implementation:

- Once a nonground IC is shown relevant we can promote the original clause, the instance as reached (maybe partially instantiated) or a particular ground instance. Promoting the entire clause may save us future potential visits to that clause but will also make relevant clause instances that are not really so and have them participate in the computation. One may opt for the middle ground as a compromise: promote the clause instance as reached during the forward relevancy computation. E.g. if the IC has the form `p(X),q(Y)--->bottom` and the partially relevant atom is `p(a)` then assert `bottom :- p(a),q(Y)` rather than, say, `bottom :- p(a),q(a)` or `bottom :- p(X),q(Y)`. However the other options are open. In the implementation the reached clause instance is promoted, be it ground or otherwise.
- Independent of the choice in the previous point, we may elect to promote to BC_i all ICs that are reachable in the forward relevancy computation or just one IC *covering* set at a time. An IC covering set is the set of ICs that terminate the fan of paths, as introduced in Section 3.1. The fan of paths only associates one clause at a choice point with each head atom; backtracking can yield other choices. The alternate choices belong to other potential IC covering sets. Once more the choice is between potentially promoting ICs not needed for the computation or being forced to redo the forward relevancy test if the promoted ones were not sufficient. Here also one may select the middle ground by promoting one IC covering set at a time while keeping the computation results in case there is a need to promote additional clauses. The implementation promotes one set at a time. Each IC set could be buffered and no IC is added to BC_i until all paths terminate in ICs (a covering set is found). If the fan of paths is not completed, no IC need be added to BC_i.
- The fact that SATCHMOREBID uses SATCHMORE as its unsatisfiability detection mechanism makes it possible to apply efficiency improvement measures that are applicable to SATCHMORE here as well. One example is the test for availability where a clause is only expanded if its atoms are both relevant and available in that they stand a chance of being proved [8)]. Another example is that a clause not allowed to be used for promoting constraints unless all of its body atoms are shown to be provable by having a link to a positive clause of the database or an element of the current interpretation.

3.4 Testing Results

Adding relevancy to SATCHMO contributed to improved performance in many cases. In cases where relevancy testing is "irrelevant", e.g. in the case where all clauses of the database are relevant for the refutation, relevancy testing is an overhead and performance will deteriorate as a result.

Similar reasoning is applicable to the introduction of forward relevancy in SATCHMOREBID. If all ICs are relevant to the refutation, any forward relevancy testing is unnecessary overhead and is bound to hurt performance. The hope is that given a query Q, combining forward and backward relevancy will isolate the clauses that need to participate in the refutation, be they ICs or otherwise.

To see how forward relevancy impacts query processing efficiency we experimented with several classes of examples. Next we report on our experiments with the SATCHMOREBID program and compare the results with those for both SATCHMO[*4] and SATCHMORE. In all cases, when the processing time [*5] exceeded 3 hours the run was aborted. The timing in this case is marked by OT.

1. **(Nonground case)** We tested variants of the problem in Example 2.5 which is borrowed from [13] with an added set of clauses that are meant to make relevant clauses that have no contribution to the refutation. Using SATCHMOREBID, the original and modified examples were processed in time comparable to that of SATCHMORE on the original (less than 0.2 seconds). SATCHMORE timing was 0.1 for the case when only the $j(g)$ fact was included, 704 seconds for the case when the $j(g), j(h)$ facts were included, and days for the case when the $j(g), j(h), j(i)$ facts were included (the computation was aborted after 3 days).
2. (**NonHorn case**) To test possible gains in performance as the proportion of irrelevant ICs increases, we used a suite of problems based on Example 2.6. We retained S_b and added a progressively expanding set of nonHorn clauses and related constraints. For each i, the tested clause set is the query negation as BC and $S_b \cup (\bigcup_{j \leq i} NH_j)$ as

[*4] SATCHMO tests were performed on a version given in [13] which is close to the SATCHMORE implementation. That version is not fair and therefore the results here represent the worst case scenario for SATCHMO as the unsatisfiable part of the theory appeared last in the clause list.

[*5] We emphasize the comparative performance of the programs on the given examples. The experimentation was done on a Sun Microsystems Ultra 1 machine, 143 MHz clock speed and 128 MB of memory .

FC_i (see Example 2.6). The results are in line with our expectations. SATCHMOREBID was able to isolate the required component for the refutation and therefore maintained a constant processing time. The performance of SATCHMORE on the other hand decreased as the proportion of irrelevant ICs increased. The results are given in column 2 of Table 1.

3. (**NearHorn case**) The examples in the previous testing have major nonHorn components. Since one may expect that real-life problems to be nearHorn [12], we performed tests on another suite of problems where the base component, S_b, is the same as that of Example 2.6 but the irrelevant components had a more nearHorn structure by having only a single nonHorn clause:

$nH_i = \{$

```
ti,ki---> ji.                ti,hi,si--->bottom.
ti,ki,ji---> mi;si;hi.       ti,ki,mi--->bottom.
si,ji---> ni.                mi,hi,ki--->bottom.
ki,mi---> ri.                ri,mi,ji--->bottom.
ti,hi--->ri.                 ji,ri,si--->bottom.
true --->ti.                 hi,si,ji--->bottom.
true --->ki.                 ni,ri--->bottom.
si,hi --->ri.                ni,mi--->bottom.}
```

We did the testing for several values of i. The clause set is as for the NonHorn case, but with $S_b \cup (\bigcup_{j \leq i} nH_j)$ as FC_i. Here too the results were in line with our expectations. SATCHMOREBID was able to isolate the required component for the refutation and therefore maintained a constant processing time. The performance of SATCHMORE on the other hand decreased as the proportion of irrelevant ICs increased, though the absolute times were always less than for the previous case, reflecting the simpler nature of the irrelevant clause set. The results are given in column 3 of Table 1.

4. (**Mixed case**) We also tested the case when the irrelevant set was a combination of the two previous cases[*6]. BC stays as before and $FC_{2i} = S_b \cup (\bigcup_{j \in \{2,4,\ldots,2i\}} (nH_{j/2} \cup NH_{j/2}))$. As expected, the results

[*6] Since the mixed case involved having a balance of nonHorn and nearHorn clauses, the numbers given are restricted to half that of the nearHorn case. Other cases are marked NA (not applicable).

were intermediate and are given in column 4 of Table 1.

Each ground test problem is balanced in the sense that the numbers of regular clauses and ICs are almost equal. This and the internal structure of the test examples explain the differences in run times between SATCHMO and SATCHMORE. However, the fact that SATCHMOREBID performs well here is an indication that it will perform well in real-life situations where excessive numbers of ICs irrelevant to the queries being considered are present such as when unrelated databases are combined. Of course, it is always possible to construct examples where relevancy testing as a whole, and forward relevancy testing in particular, are not needed and they will only add an overhead to the computation.

§4 Comparisons, Conclusions and Future Work

In this paper we presented the SATCHMOREBID approach to query answering in disjunctive databases. The approach is based on detecting relevant integrity constraints that are needed to answer the given query and having only them participate in the refutation and no other ICs. This is done through a forward search for the ICs of the database relevant to the query being processed. The detected ICs are promoted to participate in a SATCHMORE-style computation.

In as far as the effect of ICs on their clause expansion strategy, SATCHMO and SATCHMORE represent two extremes. SATCHMO is not query or IC sensitive in the sense that its clause expansion is not influenced by the ICs, while SATCHMORE is equally sensitive to all ICs: its clause expansion process is driven by ICs independent of their source. SATCHMOREBID on the other hand is query sensitive. Its expansion is driven by the query negation and only ICs shown relevant to the refutation. This approach complements the backward relevancy testing discussed in [19, 13] and implemented in SATCHMORE [19, 13] and avoids the drawback of SATCHMORE resulting from treating all ICs of the database on parity with the negation of the query.

Testing results of SATCHMOREBID showed substantially improved performance over a certain class of problems in comparison with SATCHMO and SATCHMORE. This holds for problems with a large number of ICs, and maybe a large extensional component (facts), such as the result of combining loosely coupled knowledge bases. Even for problems for which either SATCHMORE or SATCHMO are efficient, SATCHMOREBID performed well due to the ef-

ficiency of the forward relevancy check. By controlling the IC content of the initial backward chaining component, BC_0, one can manipulate the distance between SATCHMORE and SATCHMOREBID. SATCHMOREBID will simulate SATCHMORE if all ICs are included in BC_0 in which case both procedures will have almost the same performance, as was confirmed by our testing. SATCHMOREBID can be viewed as a generalization of SATCHMORE in the sense that we can achieve SATCHMORE behavior by having all negative clauses of the database as well as the query negation in BC_0.

As is the case for SATCHMORE, SATCHMOREBID is a refutational procedure rather than a model generator as is SATCHMO. In cases when the theory is satisfiable it may return only a partial model. Actually, for certain satisfiable theories the procedure may find no relevant (forward or backward) clauses and thus perform no clause expansions. SATCHMO on the other hand will return a model (of the database satisfying the query) if it fails to find a refutation and reports satisfiability. While for certain applications the model generation property is desirable, the introduction of relevancy was meant to cut short many of the computations that are needed for model generation. This pruning effect is a major source of efficiency of SATCHMOREBID.

The procedure was defined in close connection to SATCHMO and its extension SATCHMORE. However detecting relevant clauses has a more general utility and can be incorporated into other procedures, an issue that warrants pursuing as a topic of further research.

Another issue of interest is to test SATCHMO and its modifications, including SATCHMOREBID, against randomly generated sets of clauses and study the behavior of each of these systems for different classes of inputs.

Acknowledgment This research was conducted while the second author was visiting at the Computer Science Department of Duke University as a Fulbright Scholar and an AFESD Fellow. The support of the Fulbright and AFESD programs is highly appreciated.

References

1) F. Bancilhon, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, 1988.

2) F. Bry and A. Yahya. Minimal Model Generation with Positive Unit Hyper-Resolution tableaux. *Journal of Automated Reasoning*,Volume 25, Issue 1, July 2000 pages 35-82.

3) F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *Data & Knowledge Engineering*, pages 289–312, 1990.

4) C. L. Chang and K. C. Lee. *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

5) R. Demolombe. An Efficient Strategy for Non-Horn Deductive Databases. *Theoretical Computer Science*, 78:245–259, 1991.

6) T. Eiter and G. Gottlob. Complexity Aspects of Various Semantics for Disjunctive Databases, *Proceedings of the Twelfth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-93)*, 1993, June, 158-167,

7) R. Hasegawa, K. Inoue, Y. Ohta and M. Koshimura. NonHorn Magic Sets to Incorporate Top-down Inference into Bottom-up Theorem Proving. *Proceedings of CADE97* pages 176-190. 1997.

8) L. He, Y. Chao, Y. Shimajiri, H. Seki and H. Itoh. A-SATCHMORE: SATCHMORE with Availability Checking *New Generation Computing*: 16, 1998, pages 55–74.

9) C.A. Johnson. Top-down Deduction in Indefinite Deductive Databases. In *Journees Bases de Donnees Avances*, pages 119–138, Toulouse, France, 1993.

10) J. Lloyd. *Foundations of Logic Programming*. Second Edition. Springer Verlag, 1987.

11) J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.

12) D.W. Loveland. Near-Horn Prolog. In J.-L. Lassez, editor,*Proc. of the 4th Int. Conf. on Logic Programming*, pages 456–469.MIT Press, 1987.

13) D.W. Loveland, D. Reed, and D. Wilson. SATCHMORE: SATCHMO with RElevancy. *J. Automated Reasoning*, 14:349–363, July 1995.

14) R. Manthey and F. Bry. SATCHMO: a Theorem Prover Implemented in Prolog. In J.L. Lassez, editor, *Proc.* 9^{th} *CADE*, pages 456–459, 1988.

15) Y. Ohta, K. Inoue and R. Hasegawa. On the Relationship Between Non-Horn Magic Sets and Relevancy Testing. *Proceedings of CADE -15, LNAI 421*, pages 333-348, 1998.

16) D. Plaisted. An Efficient Relevance Criterion for Mechanical Theorem Proving. ,AAAI-1980, pages 79–83, 1980

17) A. Rajasekar and H. Yusuf. Dwam - A WAM Model Extension for Disjunctive Logic Programming. *Annals of Mathematics and Artifitial Intelligence*, 14:275–308, 1995.

18) R. Ramakrishnan and S. Sudarshan. Top-down vs. Bottom-up Revisited. In *Proceedings of the ISLP'91*, 1991.

19) A. Ramsay. Generating Relevant Models. In *Journal of Automated Reasoning*, Vol. 7. 1991. pages 359–368.

20) M. Stickel. Upside-down Meta-Interpretation of the Model Elimination Theorem Proving Procedure for Deduction and Abduction. *J. Automated Reasoning*, 13(2):349–363, Oct 1994.

21) A. Yahya. A Goal-driven Approach to Efficient Query Processing in Disjunctive Deductive Databases. Technical Report PMS-FB-1996-12. Department of Computer Science, Munich University. July 1996.

# of Irrelevant Blocks	NonHorn	NearHorn	Mixed
0-SATCHMORE	0.03	0.03	NA
0-SATCHMO	0.01	0.01	NA
1-SATCHMORE	0.52	0.15	NA
1-SATCHMO	0.03	0.05	NA
2-SATCHMORE	5.28	0.38	1.52
2-SATCHMO	0.13	0.12	0.23
3-SATCHMORE	45.02	0.82	NA
3-SATCHMO	4.98	0.30	NA
4-SATCHMORE	342.82	1.48	26.18
4-SATCHMO	31.58	0.65	4.45
5-SATCHMORE	2434.17	2.43	NA
5-SATCHMO	190.73	1.52	NA
6-SATCHMORE	OT	3.73	312.88
6-SATCHMO	1114.02	4.05	62.23
7-SATCHMORE	OT	5.45	NA
7-SATCHMO	OT	8.92	NA
8-SATCHMORE	OT	7.62	3065.00
8-SATCHMO	OT	21.05	797.80
9-SATCHMORE	OT	10.17	NA
9-SATCHMO	OT	46.90	NA
10-SATCHMORE	OT	13.33	26022.20
10-SATCHMO	OT	104.32	9823.69
20-SATCHMORE	OT	85.62	OT
20-SATCHMO	OT	OT	OT

Table 1 Timing (in seconds) for SATCHMORE and SATCHMO as compared to the almost constant SATCHMOREBID time of less than 0.1 seconds.