Using ACTs in Disjunctive Deductive Databases

Charlie F. Obimbo Bruce Spencer
Faculty of Computer Science,
University of New Brunswick,
Fredericton, N.B., E3B 5A3, Canada,
Email: cobimbo@unb.ca, bspencer@unb.ca.

October 11, 1996

Abstract

This paper describes research efforts to develop efficient implementation techniques for disjunctive deductive databases. The aim is to build a database engine that produces ***perfect model semantics*** as introduced by Przymusinski. When used with positive disjunctive databases the results would be equivalent to that of ***minimal model semantics***. Our strategy is to build *Access Clause Trees* (*ACTs*) based on the new data structure - *Clause Trees*, introduced by Horton and Spencer, to facilitate in building a two stage query processing engine. This engine builds the ***ACTs*** bottom up using the intensional database and then uses ***set-at-a-time*** evaluation to run the newly developed trees against the extensional database.

Keywords: *Clause trees, Access Clause Trees (ACTs), Perfect Model Semantics (PMS), merge path, priority relation, preference relation, perfect model, Disjunctive Deductive Databases (DDDB), Disjunctive Stratified Databases (DSDB), t-clause.*

1 Introduction

The fields of deductive databases and logic programming are intimately related. Theoretical developments in one area have impacted the other. This is not surprising as both subjects are an outgrowth of work in automated reasoning (*AR*). Much exciting work has been done and continues to be done in both of the areas [Min88].

The need to have efficient deductive engines to infer information from large databases cannot be overstated. Work is being done in this area, both in *AR* and in deductive databases. In *AR* the knowledge base may not be as overwhelming as in databases, but with the inclusion of functions and first-order logic, the proofs can quickly become complex.

However, until now, there are few known disjunctive deductive systems. ***DisLog***[1] and ***Toy University***[2] are some of the attempts to implement Disjunctive Deductive Databases ($DDDBs$). For a long time researchers have been working on the semantics of $DDDBs$. There is now a general agreement to use *w*ell-founded semantics, introduced by van Gelder et al [vGRS91]. Currently, one of the drawbacks of $DDDBs$ from taking their role in the real world, is the bottleneck the $DDDBs$ have on the efficiency of their inference engines.

$DDDB$ systems must compute answers according to an appropriate and accepted semantics, and do so efficiently. In this paper, we propose a system that conforms to Przymusinski's perfect model semantics . Its inference engine, based on Horton and Spencer's clause trees[3][HS95a], uses two stages. In the first stage, a set of *Access Clause Trees* is built from the intensional database. In the second, the *Access Clause Trees* are resolved against the extensional database, giving the answers.

2 Disjunctive Stratified Databases

Definition 1 [Prz88, GPP89]. Let S be the set of predicate symbols of a disjunctive normal database DB, such that S is partitioned into sets $S_1, \ldots, S_r$, called ***strata***. Let $Stratum(a) = i$ if and only if the predicate symbol of the atom a is in $Stratum(S_i)$. Then DB is called a ***disjunctive stratified database*** ($DSDB$) if for every clause[4]

$$A_1 \vee \ldots \vee A_k \leftarrow B_1, \ldots, B_m, \neg D_1, \ldots, \neg D_n \quad (k \geq 0) \tag{1}$$

of DB, there exists a constant c, $1 \leq c \leq r$, such that for all i, $1 \leq i \leq k$, $Stratum(A_i) = c$, for all j, $1 \leq j \leq m$, $Stratum(B_j) \leq c$, and for all l, $1 \leq l \leq n$, $Stratum(D_l) < c$.

The D_l's differ from B_j's in that D_l's are evaluated using ***negation as failure***, whereas the B_j's are atoms which are either facts or ***logical consequents*** of DB.

Any partitioning of S satisfying the above conditions is called a ***stratification*** of DB. In this paper, we will mainly be concerned with a class of minimal models called ***perfect models***.

---

[1] ***DisLog*** - A System for Reasoning in disjunctive deductive databases, created by ***Dietmar Seipel*** and ***Helmut Thöne*** of University of Tübingen, Tübingen, Germany - available by E-mailing to ***seipel@informatik.uni-tuebingen.de***.

[2] ***Toy University database*** created by ***PRISM Group***, under José A Fernandez and Jack Minker, University of Maryland, College Park, MD, USA, available for demonstration at ***http://karna.cs.umd.edu:3264/projects/dddbs/database.html***.

[3] ***Clause tree resolutions***, as referred to in this paper, and others referenced, differ from ***clause graph resolution*** [Kow75, Joh94], ***connection graph resolution*** [Bib81b, Bib81a], and ***model tree resolution*** [Fer91, YFM94], which is based on ***t-clauses*** [LMR92] referred to in section ***5.1*** in this paper.

[4] Note that we may at times write the clauses in clausal normal form, i.e. as a disjunction of literals, as in Example 4.1

3 Perfect Model Semantics

In *Perfect Model Semantics* (*PMS*) we assume that the clauses in DB are expressed in such a manner that their syntax conveys the intended semantics. Suppose the clauses are expressed in the form of equation (1) above, then the following conditions hold [Prz88]:

PR1: Negative premises have a higher priority than consequents
(i.e. $\forall i, l, D_l \prec A_i$).

PR2: Positive premises have priority higher than or equal to that of consequents
(i.e. $\forall i, j, B_j \preceq A_i$).

PR3: Predicates appearing in the consequents have the same priority
(i.e. $\forall C, C' \in \{A_i\}, C \preceq C'$).

Apart from these conditions, the priority relation is augmented by transitivity and closure rules:

PR4: (*transitivity of* $\preceq$) if $A \preceq B$ and $B \preceq C$ then $A \preceq C$

PR5: (*transitivity of* $\prec$) if $A \preceq B$ and $B \prec C$ (resp. $D \prec A$), then $A \prec C$ ($D \prec B$);

PR6: ($\prec$ *implies* $\preceq$) if $A \prec B$ then $A \preceq B$;

PR7: (*closure axiom*) Nothing else satisfies $\prec$ or $\preceq$.

Example 3.1 Consider a world in which the only domestic birds kept are non-flying birds. And that if an animal flies then we can either assume that it is a bird or a bat. Let us represent this as follows:

$$
\begin{aligned}
&bird(X) \lor bat(X) \leftarrow canfly(X) \land \neg ab(X).\\
&ab(X) \leftarrow domestic(X).\\
&canfly(b).
\end{aligned}
$$

Then using PMS we have:

$$
\begin{aligned}
&ab(b) \prec bird(b), ab(b) \prec bat(b) && \text{(by } PR1)\\
&canfly(b) \preceq bat(b),\ canfly(b) \preceq bird(b),\ domestic(b) \preceq ab(b) && \text{(by } PR2)\\
&bird(b) \preceq bat(b), bat(b) \preceq bird(b), && \text{(by } PR3)
\end{aligned}
$$

Consequently,

$$
domestic(b) \prec bat(b),\ domestic(b) \prec bird(b), \qquad \text{(by } PR5)
$$

Formarly we can define the priority relation as follows:

Definition 3.1 [Prz88] Let a and b be atoms in the Herbrand Base of a disjunctive stratified database DB. We say that a has a *higher priority* than b (denoted $a \prec b$) if $Stratum(a) < Stratum(b)$. The relation is called the *priority relation* between atoms.

After defining the priority relation $\prec$ between ground atoms of DB, the notion of a *perfect model* can be defined as follows:

Definition 3.2 [Prz88] Let M and N be two distinct models[5] of a disjunctive stratified database DB. We say that N is *preferable* to M ($N \prec\!\prec M$), if

$$\forall a \in (N - M) \ \ \exists b \in (M - N), \quad a \prec b$$

The relation $\prec\!\prec$ is called the *preference relation* between models. $N \preceq\!\preceq M$ if $N = M$ or $N \prec\!\prec M$. A model N of DB is said to be *perfect* if there is no model preferable to N.

Example 3.2 Let the database DB consist of the following clauses:

$$avoids\_math(X) \leftarrow businessman(X), \neg good\_mathematician(X).$$
$$businessman(perot).$$
$$physicist(einstein).$$

Then DB has two minimal models

$$M_1 = \{businessman(perot), physicist(einstein), avoids\_math(perot)\}.$$
$$M_2 = \{businessman(perot), physicist(einstein), good\_mathematician(perot)\}.$$

since the first clause may be rewritten as

$$avoids\_math(X) \vee good\_mathematician(X) \leftarrow businessman(X).$$

However, since $good\_mathematician(perot) \prec avoids\_math(perot)$, then $M_1 \prec\!\prec M_2$. Consequently, M_1 is the only perfect model of DB.

The perfect model semantics define the existential closure of C, $\exists C$, to be a logical consequence of a disjunctive stratified database DB if and only if there exists a set $\{\theta_1, ..., \theta_n\}$ of ground substitutions such that the ground positive clause $C\theta_1 \vee \ldots \vee C\theta_n$ is true in all the perfect Herbrand models of DB. For *Positive Disjunctive Deductive Databases* (negation-free), perfect Herbrand models coincide with minimal Herbrand models. For disjunctive stratified databases the set of perfect Herbrand models is a subset of the minimal Herbrand models of DB.

4 Clause Trees

4.1 An Intuitive exposition of clause trees

Horton and Spencer have come up with a new data structure, *clause trees* [HS95a], for implementing and understanding resolution. We first describe clause trees intuitively, and then go on to formally define them in section 4.2.

[5] That is the set of positive ground atoms from the *Herbrand Base* of DB.

Whilst using binary resolution, one starts with a set of ***clauses***, each of which is the disjunction of its literals, and applies resolution to them until the clause that one wants is found. This clause is the empty clause if one is looking for a contradiction [HS95b]. A clause is usually represented as a set of literals, but henceforth we represent it in graph theory terms. An input clause is usually represented by a ***clause node*** connected to ***atom nodes*** each of which is labelled by an atom. A $+(-)$ sign labels the edge joining the atom node to the clause node if the atom appears ***positively*** (***negatively***) in the clause. ***Figure 4.1(c)*** shows the tree representing the clause $\{a, b, \neg d\}$. Such a tree is called a ***clause tree*** (or an ***elementary clause tree*** since it represents an input clause). ***Figures 4.1(b)*** and (*c*) are elementary clause trees of the clauses $\{\neg b, c, \neg d\}$ and $\{d\}$ respectively.

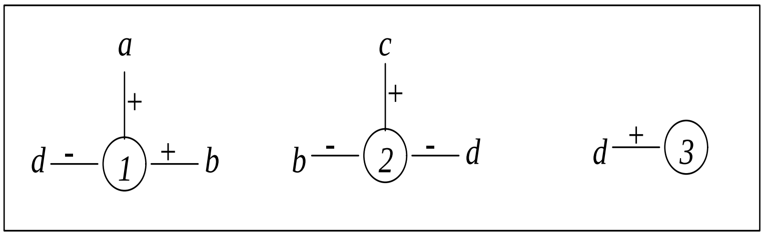

Figure 4.1

Clauses can be resolved together using resolution, if they contain complementary literals. For example, the clauses $\{a, b, \neg d\}$ and $\{\neg b, c, \neg d\}$ can be resolved on b to form the new clause $\{a, c, \neg d\}$. Clause trees are resolved by identifying the nodes that represent complementary

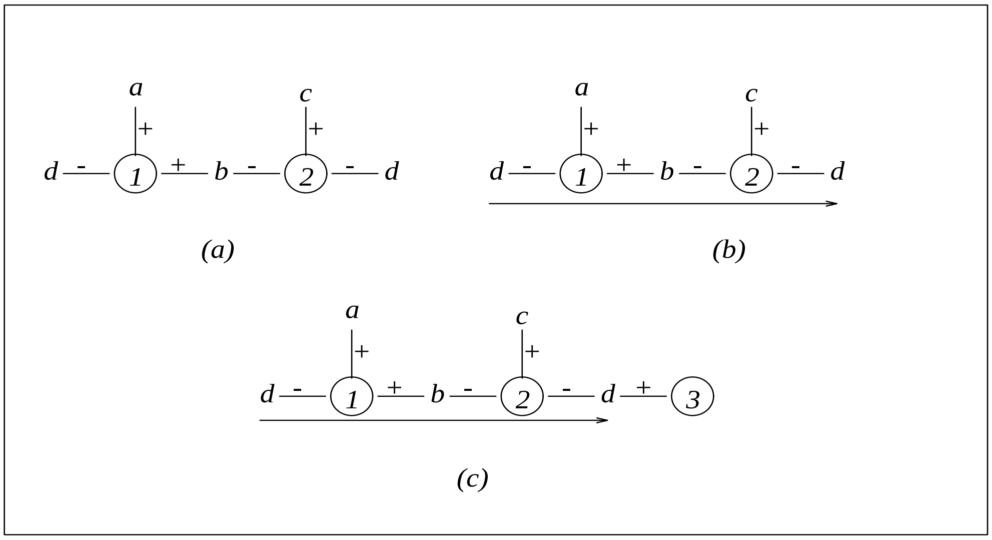

Figure 4.2

literals from two different clauses, as shown in ***Figure 4.2(a)***. The leaves of the resulting tree are the literals of the resulting clause. Albeit, two of the leaves are labelled $\neg d$ that occurs when the union of two sets occurs, is not handled automatically by the clause trees. Instead the two nodes that correspond to the same literal can be joined with a ***merge path*** as in Figure *4.2(b)*. The literal at the tail of a merge path is no longer considered to be a literal of the corresponding clause and is said to be closed in the clause tree. Finally, a resolution between Figure *4.2(b)* with the clause tree for the clause $\{d\}$ is done, resulting in the clause tree in Figure *4.2(c)* whose clause is $\{a, c\}$. The operation of resolving two clause trees followed by insertion of all leaf-to-leaf merge paths is anologous to the resolution operation.

With clause trees, the merge paths need not necessarily be added between open leaves. It may be added from an open leaf to an internal node (under the conditions given in section 4.2.) An example of this can be seen in Figure *4.3* where clause $\{\neg b, c, \neg d\}$ is resolved with $\{d\}$, and the result resolved with $\{a, b, \neg d\}$ - giving the result $\{a, c, \neg d\}$ as seen in Figure *4.3(b)*. This is the result that would be obtained when the clauses are resolved in that order - using resolution with clauses. However, inserting a merge path for d yields the clause tree in Figure *4.3(c)* $\{a, c\}$, which is the same as the one obtained in Figure *4.2(c)*. Thus a clause tree may allow us to use an internal node in the tree to remove a literal from a clause.

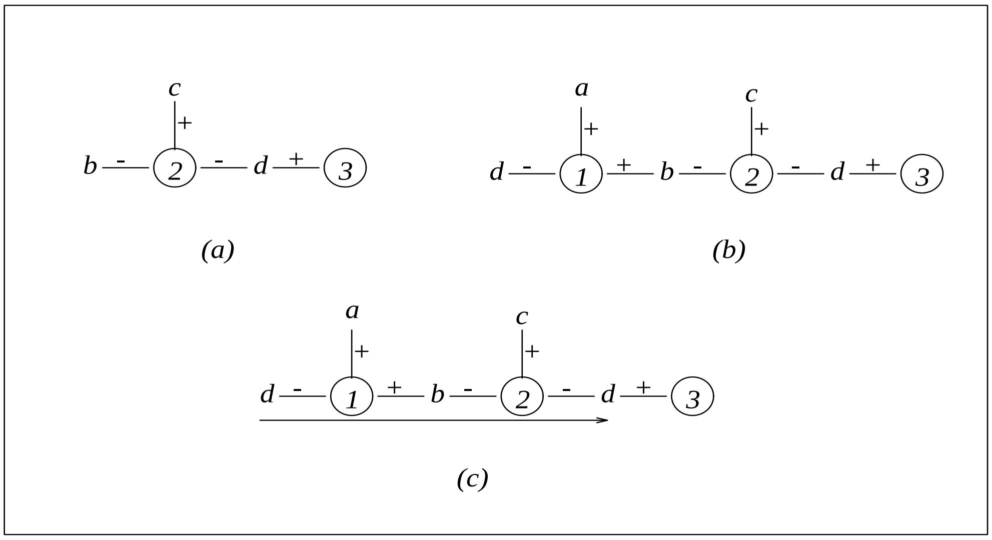

Figure 4.3

If the clause tree were much bigger, with many internal nodes, more literals might be removed from the clause. This shows one advantage of using resolution with clause trees over resolution with clauses.

We also see from above, that by using clause trees, it may not matter in which order the resolutions are done whilst searching for proofs, as the resulting clause tree can be made the same as the best result. An inference engine, therefore, that uses clause trees instead of sets to represent clauses can take advantage of this redundancy and use fewer resolutions to search for proofs than other resolution-based procedures. Moreover, since a clause really represents several distinct proofs, it is possible to detect if any of those proofs has a clause that contains a literal and its negation (or is said to contain a ***tautolgy merge path***.) Such a clause is a *tautology* and therefore cannot be useful. We can safely discontinue work on this clause tree, and hence we can cut this branch of search.

4.2 Clause tree: Definitions

More formally, we define a clause tree as follows. We first define a *path*.

Definition 4.1 [HS95a]. A **path** $path(v_0, v_n)$ in a graph from v_0 to v_n is an alternating sequence $\langle v_0 e_1 v_1 \ldots v_n \rangle$ where each v_i is a node and e_j is an edge. The first node v_0 is the *tail* of the path and the final node v_n is the *head*.

Definition 4.2 [HS95a]. $T = \langle N, E, L, M \rangle$ is a **clause tree** on a set of clauses if:

(a) $\langle N, E \rangle$ is a tree (which may be rooted or unrooted).

(b) L is a labelling of the nodes and edges of the tree. $L : N \cup E \rightarrow S^* \cup A \cup \{+, -\}$ where S^* is the set of instances of clauses in S and A is the set of instances of atoms in S. Each node is labelled either by a clause in S^* and called a clause node, or by an atom in A and called an atom node. Each edge is labeled + or -.

(c) No atom node is incident with two edges labelled the same.

(d) Each edge $e = \{a, c\}$ joins an atom node a and a clause node c; it is associated with the literal $L(e)L(a)$.

(e) For each clause node c, $L(c) = \{L(\{a, c\})L(a) \mid \{a, c\} \in E\}$. A path $\langle v_0 e_1 v_1 \ldots e_n v_n \rangle$ where $0 \le i \le n$, $v_i \in N$ and $e_j \in E$ where $1 \le j \le n$ is a *merge path* if $L(e_1)L(v_0) = L(e_n)L(v_n)$. Path $\langle v_0 \ldots v_n \rangle$ *precedes* ($\prec$) path $\langle w_0 \ldots w_m \rangle$ if $v_n = w_i$ for some $i = 1, \ldots, m-1$.

(f) M is the set of merge paths called *chosen merge paths* such that:

   (i) the tail of each is a leaf (called a *closed leaf*),
   (ii) the tails are all distinct and different from the heads, and
   (iii) the relation $\prec$ on M can be extended to a partial order.

A clause tree is said to be *closed* if all its leaves are closed. A *closed clause tree* represents an empty clause (a proof).

A set M of paths in a clause tree is *legal* if the relation on M can be extended to a partial order. A path P is *legal* in $T = \langle N, E, L, M \rangle$ if $M \cup P$ is legal. If the path joining t to h is legal in T, we say that h is visible from t. If v and u are atom nodes in a clause tree, then v *supports* u if there is a sequence of chosen merge paths $p_1, p_2, \ldots, p_k$, such that $p_1 \prec p_2 \prec \ldots \prec p_k$, v being the head of p_k and u being an interior node of p_1.

In Figure 4.4, $P_1 \prec P_2 \prec P_3$ (i.e. path P_1 precedes path P_2 which in turn precedes path P_3). Thus P_3 supports P_2, which in turn supports P_1.

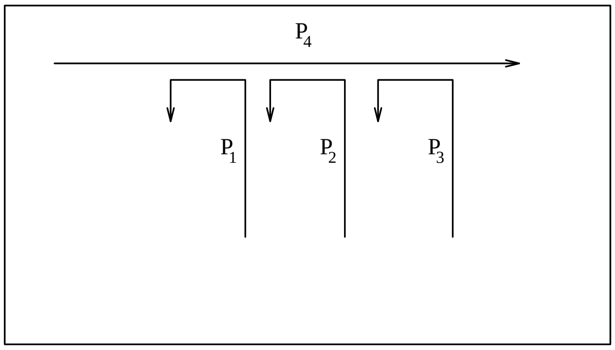

Figure 4.4 The precedence relation ($\prec$) in Clause Trees

An instance of a clause C in S is represented by an elementary clause tree which consists of a single clause node labelled by S and joined to atom nodes.

A ***minimal clause tree*** is one in which no reordering of the resolutions can produce either a merge that was not applied, or a tautology. If a merge is missed, or a tautology created in a resolution based proof, then a shorter resolution proof exists and its result subsumes the original result. In Figure 4.5, for example, (a) is a minimal clause tree, whereas (b) is not because in (b), a merge path can be drawn from one of the literals labeled c, to the other, thus resulting in a shorter proof.

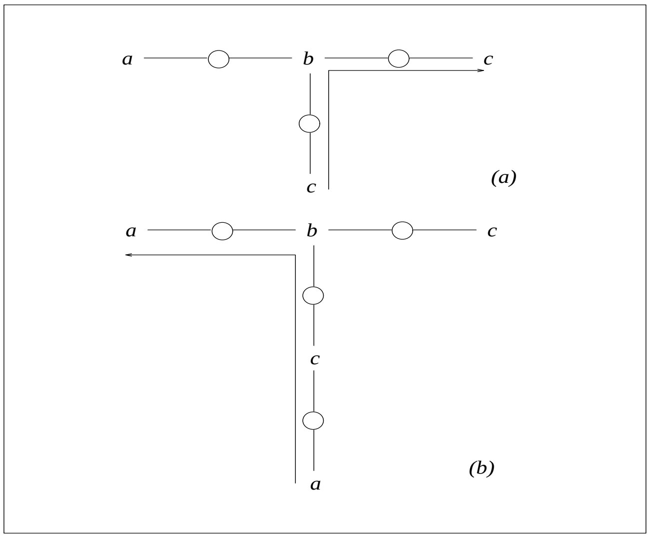

Figure 4.5 Examples of minimal and non-minimal clause trees

Thus any procedure that resolves pairs of clause trees to construct all minimal clause trees on a set of input clauses is refutationally complete for first-order logic. Three classes of such procedures have been proposed [HS95b].

Example 4.1 A complete tree for the proof of query (7) below, given clauses (1) to (6) is given in *Figure 4.6.*

1.	$\{a \vee \neg c \vee \neg d \vee \neg e\}$	5.	$\{d\}$
2.	$\{c \vee \neg d\}$	6.	$\{g\}$
3.	$\{e \vee f \vee \neg g\}$		
4.	$\{a \vee \neg f\}$	7.	$Goal : \{\neg a\}$

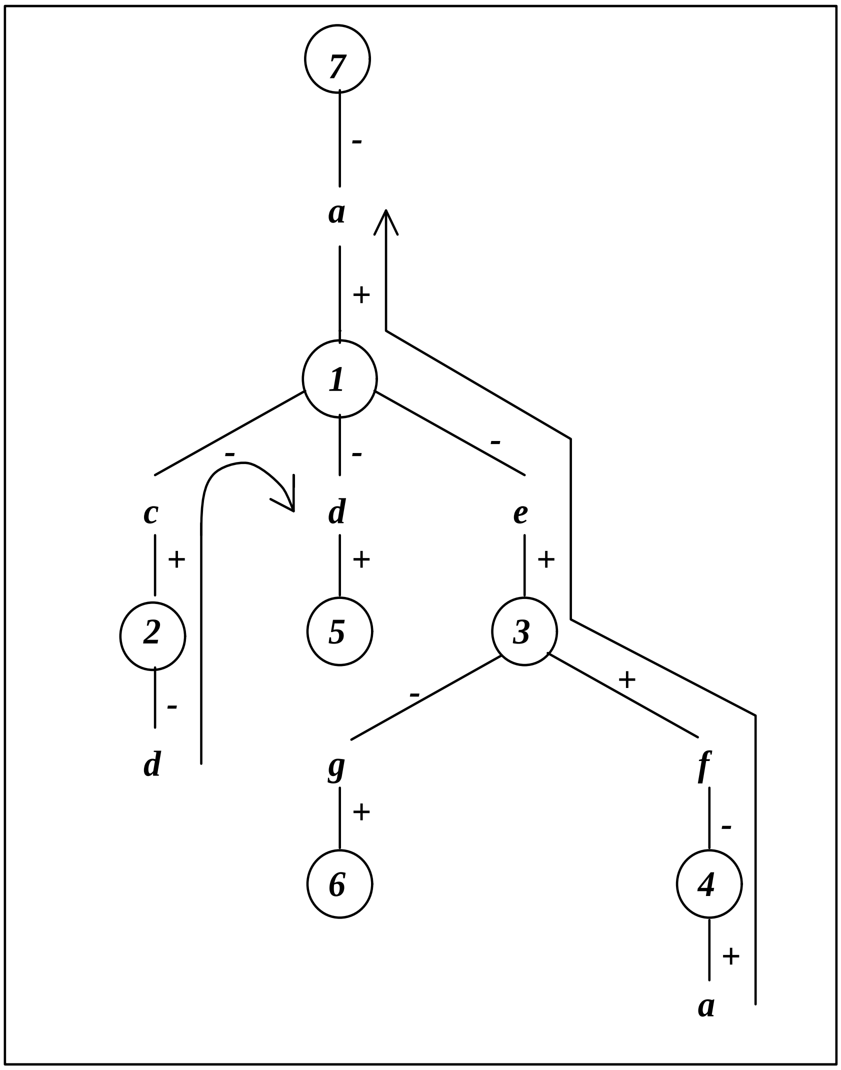

Figure 4.6 A complete clause tree for Example 4.2

5 Access Clause Trees ($ACTs$)

Before delving into the definition of Access Clause Trees ($ACTs$), we first define disjunctive databases and informally introduce $ACTs$.

Definition 5.1 [LMR92]. Let L be a function-free first order language with a finite number of constant symbols. A disjunctive logic program DB defined in L is called a ***disjunctive deductive database*** ($DDDB$). The ***intensional disjunctive database*** part of DB, $IDDB_{DB}$, is the set of program clauses in DB with at least one atom in the body of each clause. The extensional disjunctive database part of DB, $EDDB_{DB}$, is the set of clauses in DB with an empty body.

5.1 Motivation for Using $ACTs$

$ACTs$ represents a class of data structures, easy to implement and yet when used to generate a set of complete answers from a $DDDB$, its inferences are at least as efficient and in many cases more efficient than existing underlying inference mechanisms for $DDDBs$.

The following example shows how given a database, and a database query, we can use $ACTs$ to compute answers which are at least as general as those computed using ***t-clauses*** (in ***SLI resolution*** [LMR92]). Using $ACTs$ also results in fewer evaluationa of redundant answers. These answers are subsumed by other computed answer(s).

Example 5.1 Consider the following database DB:

$IDDB_{DB}$

$$
\begin{array}{rllll}
1. & u(X,Y) & \leftarrow & r(X,Y), & p_2(X,Y). \\
2. & r(X,Y) & \leftarrow & p_1(X,Y), & s(X,Y). \\
3. & p_1(X,Y) & \leftarrow & t(X,Y). & \\
4. & p_2(X,Y) & \leftarrow & s(X,Y). & \\
5. & s(X,Y) & \leftarrow & t(X,Y), & q(X). \\
6. & s(X,Y) & \leftarrow & q(X,Y), & r_2(X,Y). \\
7. & s_1(X,Y) & \leftarrow & p(X). & \\
8. & t(X,Y) & \leftarrow & s_1(X,Y). &
\end{array}
$$

$EDDB_{DB}$

$$
\begin{array}{lll}
p(1). & p(3). & p(4). \\
q(1). & q(2). & q(5). \\
r_2(1,3). & &
\end{array}
$$

Given the query *?-u(X, Y).*, SLI would use ***t-clauses*** to compute answers. In the process, the tree in ***Figure 5.1*** would be obtained. Having reached this far, it would have already built a duplicate proof for the predicate $t(X,Y)$, and would go on further to build a duplicate proof for $s(X,Y)$, since it cannot use the one to the left. This arises due to the fact that the previous $s(X,Y)$ is neither an ancestor, nor a sibling of an ancestor of the current one therefore neither t-factoring, nor t-ancestry, nor t-truncation ([LMR92] p81) can be done.

In clause trees, as can be seen in ***Figure 5.2***, both proofs for $t(X,Y)$ and $s(X,Y)$, having been computed once, are reused - by drawing a ***merge path*** from the current literal to the one on the left (i.e. to a left descendant of an ancestor), in this case, or on the right (if there is one.) The number of resolutions for the computation of a complete set of answers is therefore reduced.

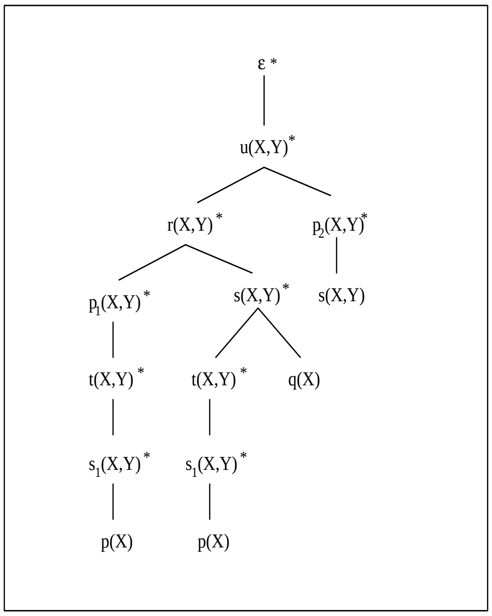

Figure 5.1 SLI evaluation to query ?-u(X, Y) on DB in Example 5.1, using t-clauses

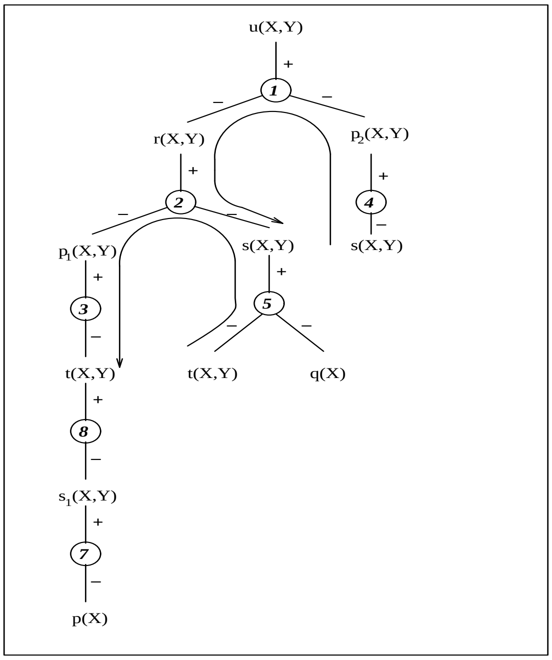

Figure 5.2 Evaluation to query ?-u(X, Y) on DB in Example 5.1, using ACTs

The answers to the query *?-u(X, Y)* for the database DB would therefore be as follows:

	SLI	*ACTs*
Clauses built before resolution with $EDDB_{DB}$	$u(X, Y) \leftarrow p(X), q(X)$. $u(X, Y) \leftarrow p(X), q(X), r_2(X, Y)$.	$u(X, Y) \leftarrow p(X), q(X)$.
Answers	$\{X = 1\}$ $\{X = 1, Y = 3\}$ $\{X = 1, Y = 3\}$	$\{X = 1\}$

Figure 5.3 Answers to query ?-u(X, Y) on DB in Example 5.1

The answers produced by $ACTs$ are at least as general as those produced by SLI, and in fact in the above example, $\{X = 1\}$ subsumes all the other answers.

Figure 5.2 is an ACT, the open leaves being $u(X, Y)$, $\neg p(X)$, and $\neg q(X)$. $u(X, Y)$ - the positive open leaf in this area, is the query asked and $\neg p(X)$, and $\neg q(X)$ - are the leaves to be resolved against the $EDDB_{DB}$.

5.2 Access Clause Trees: Definitions

We are in the process of developing a method and model that would enhance the performance of $DDDBs$. This is being done by using an *Access path*[6] which we call *Access Clause Trees* ($ACTs$). The idea is to minimize the size of the proofs for the complete set of answers to database queries, thus greatly reducing the number of inferences done, avoiding unnecessary backtracking, and also reducing the number of facts checked on.

Definition 5.2 Let DB be a disjunctive logic program, defined over a function-free first order language L. An *access clause tree* (ACT) is a minimal unrooted clause tree on the $IDDB_{DB}$.

The open leaves of an ACT thus consist of a potential query - the positive literals in the clause, and the negative literals are representative of predicates in $EDDB_{DB}$. A more general definition for $ACTs$ would include marked literals in the clauses (clause nodes) to represent negative literals of clauses with negation in the body.

In propositional logic, Figure *4.1(a)* and *(b)* and Figure *4.2 (b)* would be examples of $ACTs$. In first-order logic, an example is illustrated in *Appendix I* where a solution of the blocks and spheres database [LMR92], is given using $ACTs$.

[6] *Access path* - this is used analogously to the access path in relational databases, described as a special type of data structure that links an alternate index cluster with its associated base cluster.

6 Current Research

We are currently building a two-stage *query processing engine*, which consists of a *query pre-processor* (QP), and a *Runtime database processor* (RP). This engine produces answers from database queries according to *perfect model semantics* (PMS) on a $DSDB$. As much work as possible is done during QP leaving minimal work to the RP.

The implementation is done using the new data structure *clause trees* proposed by Horton and Spencer [HS95a]. This is done as follows:

1. Form the set T, of all clause trees from the intensional database ($IDDB_{DB}$), with only literals from the query, or extensional database ($EDDB_{DB}$) as open leaves (i.e. form all $ACTs$).

2. Form the set S of ground instances of clause trees from T using $EDDB_{DB}$.

3. Print the answers to the query.

QP implements step1, whereas steps 2 and 3 are done by RP. In-line and post-process subsumption checks are done whilst building the $ACTs$ to ensure that the QP generates only *minimal clause trees* as described in [HS95c, HS96].

The $ACTs$ are formed by using the fixed-point evaluation (*T-operator* [LMR92]) on the rules. In-line subsumption checks are done, and at this stage we ensure that all new clauses formed are checked for subsumption with already existing clauses. If they are not subsumed then the clauses are retained. This gives a tremendous boost in efficiency, in that it deters any further evaluation using subsumed clauses.

Whilst resolving predicates in function-free first-order logic, caution has to be taken when dealing with unsafe clauses. An unsafe clause is one that may resolve with itself and further resolutions would form an infinite loop.

Example 6.2 Consider the clause:

$$\{p(X,Y,Z), \neg r(Y), r(Z)\} \tag{2}$$

let us standardize it apart in preparation for resolution with say $\neg r(Y)$:

$$\{p(X1,Y1,Y), \neg r(Y1), r(Y)\} \tag{3}$$

Then when resolving (2) this with (3) we get:

$$\{p(X,Y,Z), p(X1,Y1,Y), \neg r(Y1), r(Z)\} \tag{4}$$

As can be seen, the resultant clause can be resolved with (2) resulting in a clause with more predicates, which can still be resolved with clause (2), ad infinitum. This arises due to the fact that clause (2) has a predicate and its neagation with the variables standardized apart. An example of this kind of resolution can be found in Appendix I, where $I1$ may be resolved with $I2$.

There could also be a number of clauses having the same effect when resolved one with another. These form an *unsafe* set of clauses. Such clauses may lead to the formation of an infinite number of clauses, hence the need arises to tackle it, say, by use of *iterative deepening*, the depth being determined by the *extensional database*.

In stage two of the implementation, the resulting *ACTs* are then used by *RP* to do *set-at-a-time* evaluation of answers with ground clauses. Definite ground clauses are treated separately from indefinite ground clauses, and clauses with negation in the body are also treated separately.

We further hope to investigate the possibility of incorporating the model within a relational database, i.e. whether it will be possible to have stratification and prioritisation using a relational database as an extensional database.

7 Conclusion

In this paper, we have introduced a new structure, a subclass of *clause trees* with open leaves, called *Access Clause Trees* (*ACTs*). We discussed how they can be used to build an inference engine that implements *Perfect Model Semantics* on *Disjunctive Stratified Databases*. We showed the simplicity of representation of information using *ACTs*, and *clause trees* in general.

We discussed how our engine is built using bottom-up methods for evaluation of *ACTs* using the intensional part of the database, and then using *set-at-a-time* processing to form the ground instances of the *ACTs* from the extensional database.

We further discussed some techniques that can be used to make the database query engine more efficient.

References

[Bib81a] W. Bibel. On matrices with connections. *J.ACM*, 28:633–645, 1981.

[Bib81b] W. Bibel. On the completeness of connection graph resolution. In *Proc. GWAI-81, Springer Informatik, Fachberichte, vol 47, J. H. Siekmann (ed.) 246-247*, 1981.

[Fer91] J. A. Fernandez. Bottom-up evaluation of hierarchical disjunctive deductive databases. In *in Koichi Furukawa (ed.). Logic Programming Proceedings of the Eigth International Conference, MIT Press, 660-675*, 1991.

[GPP89] M. Gelfond, H. Przymusinska, and T. C. Przymusinski. On the relationship between circumscription and negation as failure. *AI*, 38:75–94, 1989.

[HS95a] J. D. Horton and E. B. Spencer. Clause trees: A tool for implementing and understanding automated reasoning. *TR95-095, Faculty of Computer Science, University of New Brunswick, June 1995*, 1995. Also available at http://www.cs.unb.ca/profs/bspencer/htm/clause-trees/ TR95-095.ps.Z.

[HS95b] J. D. Horton and E. B. Spencer. Reducing search with minimal clause trees. Technical report, TR95-099, Faculty of Computer Science, University of New Brunswick. October, 1995, 1995. Available at http://www.cs.unb.ca/profs/bspencer/htm/clause-trees/ TR95-099.ps.Z.

[HS95c] J. D. Horton and E. B. Spencer. A top down algorithm to find only minimal clause trees. *In Dreschler-Fisher L. and Pribbennow S. eds, KI-95 Activities, Proceedings of 19th Annual German Conf. on AI, Bielefeld, Sept 11-13.*, pages 79–80, 1995.

[HS96] J. D. Horton and E. B. Spencer. Bottom-up procedures for minimal clause trees. in preperation, 1996.

[Joh94] C. A. Johnson. On the termination of clause graph resolution. *Journal of Automated Reasoning*, 13:1:83–115, 1994.

[Kow75] R. Kowalski. A proof procedure using connection graphs. *J.ACM*, 22:572–595, 1975.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* The MIT Press Cambridge, Massachusetts., 1992.

[Min88] J. Minker. Perspectives in deductive databases. *Journal of Logic Programming*, 5:33–60, 1988.

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Pub., Washington, D. C., 1988.

[vGRS91] A. van Gelder, K. A. Ross, and J. S. Schipf. The well-founded semantics for general logic programs. *JACM*, 38:3:620–650, July, 1991.

[YFM94] A. Yahya, J. A. Fernandez, and J. Minker. Ordered model trees: A normal form for disjunctive deductive databases. *Journal of Automated Reasoning*, 13:177–143, 1994.

Appendix I

Example A.1 [LMR92] Let *DB* be a disjunctive deductive database describing a world of blocks and spheres shown in *Figure A.1*.

The $EDDB_{\mathrm{DB}}$ consists of:

$E1$: blk(1).	$E5$: red(1).
$E2$: blk(2) ∨ sph(2).	$E6$: blue(3).
$E3$: blk(3).	$E7$: blue(4).
$E4$: sph(4).	$E8$: top(1,2).
	$E9$: top(2,3).

The $IDDB_{DB}$ consists of:

$I1$: red(X) ∨ blue(X) ← blk(X).
$I2$: redblue(X,Y) ← top(X,Y), red(X), blue(Y).
$I3$: blk(X) ← top(X,Y).

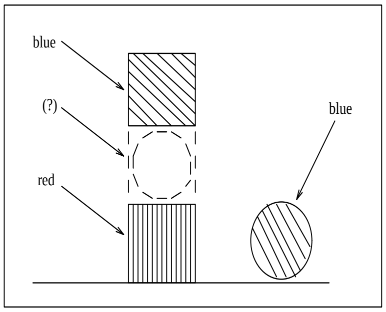

Figure A.1 The world of Blocks and Spheres of Example A.1

The clause tree for the query *redblue*$(X,\ Y)$ is shown in *Figure A.2.*

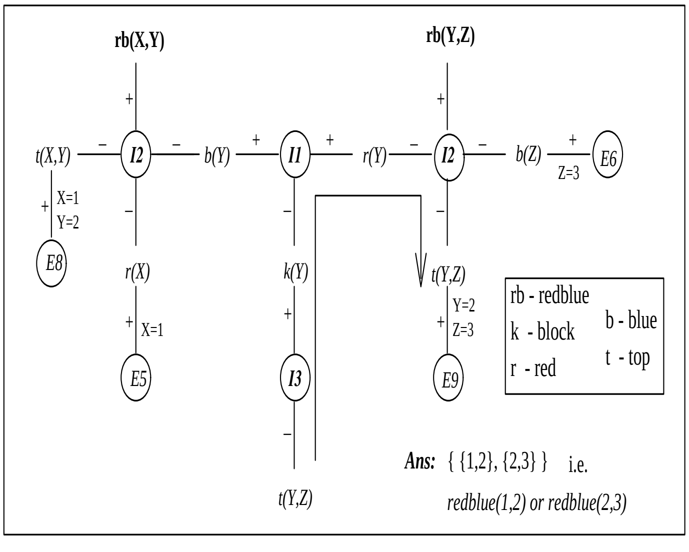

Figure A.2 Computing the query redblue(X, Y)