DOCUMENT : impress/2/01/1

ISSUE : 01

DATE : September 12, 1997

PREPARED BY : M.M.West, C.H.Bryant

THE IMPRESS PROJECT

Assigning Blame to General Clausal Form Theories

School of Computing and Mathematics,
The University of Huddersfield

Contents

1 Introduction

This report describes progress during Step 3 of the IMPRESS[1]project and should be read in conjunction with [Bry97]. IMPRESS builds on a previous project, FAROAS, which utilised an environment, the FREE, to validate a specification in MSFOL. (See [MPN+95, WB96].) An aim of IMPRESS is to enhance the existing FREE environment by utilising methods from Machine Learning (ML) such as Explanation-Based Generalisation (EBG) and Theory Revision (TR).

The domain considered by both FAROAS and IMPRESS is that of the movement of air traffic in the eastern half of the North Atlantic ('Shanwick'). The specification under consideration is a 'conflict prediction specification' (CPS) for the control of aircraft flying in the Shanwick area. The requirements, written in many sorted logic, consists of a theory of over 1,000 axioms, held in a tools environment supporting validation. Two of the tools in the environment were a parser for identifying syntactic errors, and a executable form generator (translating the CPS into Prolog). Batches of expert-derived test cases were used to compare expected and actual results. A test took the form of two flight plans in conflict violation with one-another, or else separated to the required standard. Other validation strategies included reasoning about the CPS's internal consistency and producing a 'Validation Form' of the CPS written in structured natural language. Each of the validation strategies uncovered errors in the initial encoding of the requirements, and their use improved the accuracy of the model. However, tests may succeed for the wrong reasons, and where tests fail (i.e the expert decision is at variance with the prototype's decision) it is still very difficult to identify the faulty or incomplete requirements.

Theory revision tools take an existing first order logic theory (in pure Prolog for example) and a test (example) set as input. A revised version of the theory is output which will entail the examples. However existing tools, such as [RM95], do not accept as input theories containing negation. This severely restricts the functionality of the revision tool because many theories contain negation, including the specification under consideration; one of the aims of the work described in this report is to enable a revision tool to be built which accepts negation.

The current version of the CPS has been translated to Sicstus Prolog, and the translation gives rise to predicates corresponding to main axioms, and those (definitional) corresponding to auxiliary axioms and domain objects. Both main and auxiliary contain *general* predicates, which allow negative literals in their clause bodies. The executable form of the CPS is complex, containing 50 clauses arising from main axioms, 250 auxiliary and domain object clauses, and over 1000 facts concerned with aircraft, airfields and flight plans.

The work described in this report largely involves 'blame assignment', applied to the clauses of the executable form of the CPS. This means assessing the contribution of clauses to a *misclassified instance*. In particular our interest is in a *false* positive instance. This is the case when execution of flight plans information for two aircraft results in a diagnosis of conflict violation by the tool. However National Air Traffic Control Services (NATS) experts have previously classified the flight plans as conflict free.

The IMPRESS theory revision program utilises blame assignment as a method of uncovering

[1]IMProving the quality of formal REquirements SpecificationS

errors in the CPS. However this strategy will not uncover all the errors in a program. A query that should succeed may do so but for the wrong reasons. Similarly a query that should fail may do so but for the wrong reasons. This report also describes a method which identifies *redundant* clauses in a program, viz clauses which are never used in a given set of instances.

We describe the concept of blame assignment in Section 2; we also describe the processing of the CPS which is necessary to implement it and the representation of a proof in the form of a proof tree via an established 'meta-interpreter'. However the 'text-book' meta-interpreters for generating proof trees [Bra90, SS94] are restricted to definite programs; they cannot cope with general programs, that is programs which include negative literals. Previous research, as in [Pet92, FT94] has not extended meta-interpreters for generating proof trees to include general programs. In Section 3 we describe a meta-interpreter for generating proof trees which explicitly represent *negative* clauses. Section 4 presents results utilising data provided by NATS and Section 5 describes a method of discovering redundant clauses. Appendices A – C provide some theoretical background and Appendix D describes previous work. Appendix E presents the program code for the blame assignment Algorithm, and Appendix F presents the code for discovering redundant clauses.

The work described in this report has previously been presented at LOPSTR '97 [WBM97] and as far as these authors are aware is a new contribution to the field.

2 Blame Assignment

2.1 'Shield' Status for Clauses

An assignment of blame is made in order to decide which clauses are candidates for revision. However not all clauses are subject to revision, some clauses are 'shielded' from blame assignment. This means that a proof tree for an instance need not have the proofs of all its clauses in the tree. The concept of shielding is hierarchical, thus those clauses of which the predicate is the head are either all shielded or all unshielded. In the work described here, shielded predicates are

1. 'definitional' predicates, whose proof is not required. (We assume a hierarchy where shielded predicates have only clauses of shielded predicates in their bodies.) For a given predicate, clauses associated with it are either *all* shielded or *all* unshielded. For the CPS the shielded predicates are derived from auxiliary or domain axioms.
2. predicates 'built-in' to Prolog, such as 'is', '<', etc.

2.2 'Enveloping' the CPS Code

In order to produce a robust version of the tree generator, information about the head and body of every clause in the 'theory plus background' is stored in a Prolog structure. Each clause is provided with an automatically generated identity number and the information as to its shielded 'status'. This is for efficiency and convenience as there may be many clauses associated with a given predicate. The enveloping has the following form. For each clause in the program to be revised, there is an enveloped equivalent, consisting of the identity number of each clause, together with the clauses themselves. 'Facts' are provided with a body 'true'. Undefined predicates are provided with a body 'fail' (see Appendix B). A template and examples are provided for illustration.

```
/* Template */
Head :- Body.

/*enveloped version -  Shield_status is either shielded or unshielded */
clause_impress(Head, Body, Clause_Id, Shield_status),

/* Examples */

/* Example of unshielded rule */
profiles_are_in_oceanic_conflict(Segment1,Profile1,Segment2,Profile2):-
    Segment1 belongs_to Profile1,
    Segment2 belongs_to Profile2,
    segments_are_in_oceanic_conflict(Segment1,Segment2),
    !.

/* Enveloped version of unshielded rule */
clause_impress(
 profiles_are_in_oceanic_conflict(Segment1,Profile1,Segment2,Profile2),
   (Segment1 belongs_to Profile1,
    Segment2 belongs_to Profile2,
    segments_are_in_oceanic_conflict(Segment1,Segment2),
    !)
 ,1,unshielded).

/* Example of shielded rule */
both_are_flown_in_level_flight(Segment1,Segment2):-
    is_flown_in_level_flight(Segment1),
    is_flown_in_level_flight(Segment2),
    !.

/* Enveloped version of shielded rule */
clause_impress( both_are_flown_in_level_flight(Segment1,Segment2),
 (is_flown_in_level_flight(Segment1),is_flown_in_level_flight(Segment2),!),
1001,shielded).

/* Example of shielded fact */
the_Aircraft_on_profile(profile_BWA901_1,bwa901).

/* Enveloped version of shielded fact*/
clause_impress(
the_Aircraft_on_profile(profile_BWA901_1,bwa901), true, 100169,shielded).
      shielded).
```

If a clause whose identity number is Id fails, (i.e. its negation succeeds) then it is provided with a new identity, $-Id$, in the proof tree, as in the example output of Section 3.4

2.3 Algorithm

In order to decide which clauses in a program contribute most to the proof of a false positive instance, a proof is generated for each instance and blame assigned as follows:

```
1. For each instance

        generate a generalised proof tree
        form list of unshielded rules which does not include facts
        form list of IDs for these rules
        remove duplicates from this list

2. Append together all the uniqued ID-lists for the instances and sort
the resulting list into ascending order.

3. Remove duplicates from the appended list, annotating it with counts.
```

3 Proof Trees

The generation of proof trees in logic programs is usually accomplished via *meta-interpreters*. Meta-interpreters are capable of analysing, interpreting and transforming other programs, which they treat as data [SS94]. Applications of meta-interpreters therefore include proof tree generation for theory revision and also for program debugging. In order for program debugging it is necessary for a proof tree to be generated which captures the *generalised* proof of the instance as well as one which captures the proof of the instance. The meta-interpreter utilised for proof tree generation is based on techniques from *Explanation Based Generalisation* (EBG).

3.1 Explanation Based Generalisation

The code which generates a proof tree from a proof is a robust version of that presented in [KCM87]. The concept of EBG is as follows:

Given:

- a target concept, that is the concept being learned;
- a training instance of the target concept;
- a domain theory, that is a set of facts and rules about the domain;
- an operationality criterion, specifying the form of the learned concept definition.

Determine:
A generalisation of the training instance that is a sufficient condition of the target concept and that satisfies the operationality criterion.

The concept of proof tree generators in theory revision can be similarly expressed:

Given:

- a theory, that is subject to revision;
- a 'false positive' instance of the target theory;
- a domain theory, that is a set of facts and rules about the domain;
- an assignment of shielding status to each clause.

Determine:

A generalisation of the false positive instance that is a sufficient condition of the revisable theory, where some clauses are shielded from the proof.

3.2 Example

We will use the following program to illustrate the different trees obtained through proof tree generation. For reasons which will be explained, each clause is numbered (and all are unshielded).

```
\* clauses numbered from 101 to 104 *\
r(A) :- t(A), not(p(A,Y)).

p(A, 2) :- m(A, X), Y is 12*X, Y < 24000 .
p(A, 3) :- m(A, X), Y is 12*X, Y < 36000 .
p(A, 1) :- m(A, X), Y is 12*X, Y < 20000 .


\* clauses numbered from 111 to 114 *\
m(a, 1000). m(b, 3000). t(a). t(b).
```

3.3 Tree Generation with Negative Literals Shielded

This section describes the algorithm utilised for tree generation and based on 'traditional' EBG. The following is an example of a proof tree generated from the code in 3.2. The first argument is the 'instance', viz. `r(b)`, and the second the generalisation of it. The third argument provides the (output) proof of the instance and the fourth the *generalisation* of the proof of the instance. (This provides a means of analysing the original code.) The identity number of the clause is also provided, where 'not' is given the identity of 'built in' predicates, viz. zero.

```
| ?- gen_trees(r(b), r(W), P, GenP).

P = [101,[r(b),[114,t(b), 0,not(p(b,_A))]]],
GenP = [101,[r(W),[114,t(W), 0,not(p(W,_B))]]] ?
```

A formal version of the tree and generator is presented in a manner which is independent of the computation rule and is contained in Appendix C.1. The version given here depends on Prolog's leftmost first computation rule.

The tree generator is of the form:

```
gen_trees( +Expr, :Gen_Expr, ?P, ?GenP).
```

where `Expr` can take the form:

1. `Atom`: a positive literal;
2. `(Expr, Expr)`: conjunction;
3. `(Expr ; Expr)`: disjunction;
4. `not(Expr)`: negated expression.

`Gen_Expr` is a generalisation of `Expr`. The algorithm for generating proof trees is presented now, on a case by case basis of `Expr`.

3.3.1 Case 1 – Expr a Positive Literal

When `Expr` is a positive literal, `Head`, there are several possibilities, depending on whether `Head` matches the head of an unshielded clause in the program, or a shielded clause. In every case Prolog interpretation accompanies meta interpretation. For example for `Head` unshielded, `Body` is called to check that it succeeds. This is necessary because `Head` may match the head of a Prolog procedure rather than just a single rule. If `Body` fails then Prolog will backtrack to the other rules in the procedure.

```
/* Head is unshielded */
gen_trees(Head, Gen_head, [Clause_ID_no, Proof],
                           [Clause_ID_no, Gen_proof] ) :-

        clause_impress(Gen_head, Gen_body, Clause_ID_no, Shield_status),
        copy( (Gen_head:-Gen_body), (Head:-Body) ),
        Body,

        gen_trees(Body, Gen_body, Body_proof, Gen_body_proof),
        append([Head],[Body_proof], Proof),
        append([Gen_head], [Gen_body_proof], Gen_proof).

/* Head is shielded */
```

```
gen_trees(Head, Gen_head, [Clause_ID_no, Head], [Clause_ID_no, Gen_head,]) :-

        clause_impress(Gen_head, Gen_body, Clause_ID_no, shielded),
        copy( (Gen_head:-Gen_body), (Head:-Body) ),
        Body.

/* Head is a fact */

 gen_trees(Head,Gen_head,[Clause_ID_no, Head],[Clause_ID_no,
                          Gen_head] ]):-
        clause_impress(Head,true,Clause_ID_no,_).

/*Head is 'built-in' predicate, A */

gen_trees(A, Gen_A, [0, A], [0, Gen_A], [0]) :-
        built_in(A),
        A.
```

3.3.2 Case 2 – Expr a Conjunction

This concerns the case where `Expr` is a conjunction of two expressions `A,B`. In that case the two proofs are appended.

```
gen_trees((A, B), (Gen_A, Gen_B), Proof, Gen_proof,GenIds) :-

        gen_trees(A, Gen_A, A_Proof, Gen_A_Proof, Gen_A_Ids),
        gen_trees(B, Gen_B, B_Proof, Gen_B_Proof, Gen_B_Ids),
        append(A_Proof, B_Proof, Proof),
        append(Gen_A_Proof, Gen_B_Proof, Gen_proof),
        append(Gen_A_Ids, Gen_B_Ids, GenIds) .
```

3.3.3 Case 3 – Expr a Disjunction

This concerns the case where `Expr` is a disjunction of two expressions, viz. `(A ; B)`. In that case `A` is tested first and if it fails, `B` is tested. Thus *two* trees are possible.

```
gen_trees((A ; _), (Gen_A ; _), Proof, Gen_proof, GenIds) :-
        gen_trees(A, Gen_A, Proof, Gen_proof, GenIds).

gen_trees((_ ; A), (_ ; Gen_A), Proof, Gen_proof, GenIds) :-

        gen_trees(A, Gen_A, Proof, Gen_proof,GenIds).
```

3.3.4 Case 4 – Expr Negated and Treated as Built In

```
/* For the 'traditional' EBG tree, negated expressions (Case 4) are treated as
built in.
*/
gen_trees(A, Gen_A, [0, A], [0, Gen_A], [0]) :-
        built_in(A),
        A.
```

3.4 Tree Generation Including Expanded Negative Literals

The tree generated in Section 3.3 did not contain information about the negative literals. Thus there is no information about the success of `not(p(b,_A))`. The tree generator presented in this section is a way of solving this problem, where negated clauses are expanded via De Morgan's Laws. Thus given the example code presented in Section 3.2 above, a call to the extended tree generator will provide the following response:

```
| ?- gen_trees(r(b), r(W), P, GenP).

P = [101,[r(b),[114,t(b),
        -102,[not(p(b,2)),[112,m(b,3000),0,36000 is
           12*3000,0,not(36000<24000)]],
        -103,[not(p(b,3)),[112,m(b,3000),0,36000 is
           12*3000,0,not(36000<36000)]],
        -104,[not(p(b,1)),[112,m(b,3000),0,36000 is
           12*3000,0,not(36000<20000)]]]]],
GenP = [101,[r(W),[114,t(W),
           -102,[not(p(W,2)),[112,m(W,_B),0,
              _A is 12*_B,0,not(_A<24000)]],
           -103,[not(p(W,3)),[112,m(W,_D),0,
              _C is 12*_D,0,not(_C<36000)]],
           -104,[not(p(W,1)),[112,m(W,_F),0,
              _E is 12*_F,0,not(_E<20000)]]]]] ?
```

As can be seen each of the negated clauses is expanded out and is represented in both the proof of the instance and in the generalised proof (for code analysis). *Negated* clauses are provided with a negated identity number. Since there are three clauses with predicate head p, all contribute to the proof. Since clauses `m(b,3000)`, `36000 is 12*3000` all succeed and contribute to the proof, they are included. The tree generator is of the form

```
gen_trees( +Expr, :Gen_Exp, ?P, ?GenP).
```

where `Expr` can take the form:

1. `Atom`;

2. `(Expr, Expr)`;

3. `(Atom ; Expr)`;

4a. `not(Literal)`;

5. `not(Expr, Expr)`: negated conjunction;

6. `not(Expr ; Expr)`: negated disjunction;

where Cases 4a, 5 and 6 replace 4 and extend the tree generator by expanding out negated terms using De Morgan's laws and (where applicable) by replacing a clause head with its body. The extension to the tree is as follows:

3.4.1 Case 4a – Expr Negated Literal, Unfolding

For `not(Literal)`, where `Literal` is a positive atom which matches the head of unshielded clause `A :- B` *unfolding* of the clause takes place and `not(A)` is replaced by `not(B)`. However there may be many clause heads matching `Literal`, and as *each* has to fail for `not(Literal)` to succeed, all must be included in the proof tree. (See Appendix C.2.)

```
/* The code generates a set of clause ids whose heads match Literal.
*/

gen_trees(not(A), not(Gen_A) , Set_of_Heads, Set_of_GenHeads) :-

     copy_term(Gen_A, Copy_Gen_A),
     setof(N,
       G_B^ clause_impress( Copy_Gen_A, G_B, N , unshielded) ,
                                Set_of_Ids),
      gen_conjoined_set( not(A), Gen_A, Set_of_Ids, Set_of_Heads,
                          Set_of_GenHeads).
```

the `setof` predicate is a Prolog device for obtaining a set of objects satisfying a given predicate. In this case the set of objects satisfy:

```
 G_B^ clause_impress( Copy_Gen_A, G_B, N , unshielded)
```

which, roughly translated means

```
there exists  G_B where clause_impress( Copy_Gen_A, G_B, N , unshielded)
```

The predicate `gen_conjoined_set` produces a conjoined set of proofs of `not(A)`, and also of `not(Gen_A)`. An example is provided by the proofs of `not(p(A,Y))` in Section 3.4.

```
/* Body of unfolded clause is itself shielded */

gen_trees(not(A), not(Gen_A) ,
        [Neg_Head_Id, [not(A),[Neg_Bod_Id, B]]],
        [Neg_Head_Id,  [not(Gen_A), [Neg_Bod_Id,Gen_B ]]]) :-
```

```
        clause_impress(Gen_A, Gen_B, Head_Id, unshielded),

        Neg_Head_Id is -Head_Id,
        copy( (Gen_A:-Gen_B), (A:-B) ),
        \+(B),
        clause_impress( Gen_B, _, Bod_Id, shielded),

        Neg_Bod_Id is -Bod_Id.
```

When `Literal` matches a shielded clause head there is no unfolding. If `Literal` is undefined, it fails.

```
gen_trees(not(A), not(Gen_A), [Neg_Clause_Id, not(A)],
                             [Neg_Clause_Id, not(Gen_A)]) :-

         \+(A),
         clause_impress(Gen_A, _,  Clause_Id, shielded),

         Neg_Clause_Id is -Clause_Id.

/*   predicate is  undefined */

gen_trees(not(Head) ,not(Gen_head),[Neg_Clause_ID_no,
               not(Head)],[Neg_Clause_ID_no, not(Gen_head)]):-

         \+(Head),
         clause_impress(Gen_head, fail, Clause_ID_no,_),
             Neg_Clause_ID_no is -Clause_ID_no.
```

For `not(Literal)`, where `Literal` is `not(B)`, a negated literal, `not(not((B))` is replaced by `B`.

```
gen_trees(not(A), not(Gen_A) ,  Proof,  Gen_proof) :-
        (A = not(B)),
         gen_trees(B, Gen_B ,  Proof,  Gen_proof, GenIds).
```

3.4.2 Case 5 – Expr Negated Conjunction, Expansion

A negated conjunction `not(Expr1, Expr2)` is expanded into a disjunction `not(Expr1); not(Expr2)` using De Morgan's laws. Recalling that `not(Expr1, Expr2)` means $\neg\ \exists\mathbf{y}(\mathit{Expr}1 \wedge \mathit{Expr}2)$, where $\mathbf{y}$ are the uninstantiated variables of $(\mathit{Expr}1 \wedge \mathit{Expr}2)$, then

$$
\begin{aligned}
&\neg\ \exists\mathbf{y}\ (\mathit{Expr}1 \wedge \mathit{Expr}2) \\
&\quad \longleftrightarrow \forall\mathbf{y}\neg\ (\mathit{Expr}1 \wedge \mathit{Expr}2) && \text{[Quantifier property]} \\
&\quad \longleftrightarrow \forall\mathbf{y}(\ \neg\ (\mathit{Expr}1) \vee \neg\ (\mathit{Expr}2)\ ). && \text{[De Morgan]}
\end{aligned}
$$

However $Expr1, Expr2$ may share variables; we then utilise the logical equivalence

$$\neg A \vee \neg B \leftrightarrow (A \wedge \neg B) \vee \neg A.$$

Thus if `not(Expr1)` fails, then `Expr1` is computed so that any variables it shares with `(Expr2)` can be instantiated. This is to prevent floundering of `not(Expr2)`. (For a fuller explanation, see Lemma 1, Appendix C.2.) For this reason, the proof of `Expr1` is appended to that of `not(Expr2)`.

```
gen_trees(not((A , _)), not((Gen_A ,_) ), Proof, Gen_proof) :-

        not(A),
        gen_trees(not(A), not(Gen_A), Proof, Gen_proof).

gen_trees(not((A , B)), not((Gen_A , Gen_B)), Proof, Gen_proof) :-
        A, not(B),
        gen_trees( A, Gen_A , A_Proof, Gen_A_proof),
        gen_trees(not(B), not(Gen_B), Neg_B_Proof,
                   Neg_B_Gen_proof),
        append(A_Proof, Neg_B_Proof, Proof),
        append(Gen_A_proof, Neg_B_Gen_proof, Gen_proof).
```

3.4.3 Case 6 – Expr Negated Disjunction, Expanded

A negated disjunction `not(Expr1; Expr2)` is expanded into a conjunction `not(Expr1), not(Expr2)` using De Morgan's laws. The proofs are appended, as in Case 2.

```
gen_trees(not((A ; B)), not((Gen_A ; Gen_B)), Proof, Gen_proof) :-

                not(A),
                not(B),
                gen_trees(not(A), not(Gen_A), A_Proof, Gen_A_Proof),
                gen_trees(not(B), not(Gen_B), B_Proof, Gen_B_Proof),
                append(A_Proof, B_Proof, Proof),
                append(Gen_A_Proof, Gen_B_Proof, Gen_proof).
```

4 Blame Assignment - Tests

4.1 Test Data

The test data for the blame assignment was obtained from NATS in its raw form and consisted of a days cleared flight plans (03/01/1995 –04/01/1995). Since all the plans were cleared they provide us with *negative* instances, viz pairs of aircraft which are not in conflict violation as classified by NATS experts. The raw data was translated to MSFOL, and subsequently (by our validation tool) to its executable form in Prolog. Thus each aircraft flight profile is modelled by an associated set of 'facts' in Prolog. The example below shows the flight profile and other relevant data for GAF033 in MSFOL and the corresponding Prolog facts.

Flight Profile Example in MSFOL

```
"the_Aircraft_on(profile_GAF033_1) = GAF033".

"the_Type_of(GAF033) = EA31".

"(GAF033 meets_mnps)".

"(the_Segment(profile_GAF033_1, 54 20 N ; 012 00 W ; FL 330 ; FL 330 ;
 10 58 GMT day 0, 56 N ; 020 W ; FL 330 ; FL 330 ; 11 35 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".

"(the_Segment(profile_GAF033_1, 56 N ; 020 W ; FL 330 ; FL 330 ;
 11 35 GMT day 0, 57 N ; 030 W ; FL 330 ; FL 330 ; 12 19 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".

"(the_Segment(profile_GAF033_1, 57 N ; 030 W ; FL 330 ; FL 330 ;
12 19 GMT day 0, 57 N ; 040 W ; FL 330 ; FL 330 ; 13 01 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".

"(the_Segment(profile_GAF033_1, 57 N ; 040 W ; FL 330 ; FL 330 ;
 13 01 GMT day 0, 56 N ; 050 W ; FL 330 ; FL 330 ; 13 44 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".

"(the_Segment(profile_GAF033_1, 56 N ; 050 W ; FL 330 ; FL 330 ; 13 44
GMT day 0, 54 37 N ; 055 52 W ; FL 330 ; FL 330 ; 14 12 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".

"(the_Segment(profile_GAF033_1, 54 37 N ; 055 52 W ; FL 330 ; FL 330 ;
14 12 GMT day 0, 53 37 N ; 058 08 W ; FL 330 ; FL 330 ; 14 25 GMT day 0, 0.80)
 belongs_to profile_GAF033_1)".
```

Flight Profile Example – Prolog Facts

```
the_Aircraft_on_profile(profile_GAF033_1,gaf033).

the_Type_of(gaf033,ea31).

meets_mnps(gaf033).

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(54,20),
long_W(12,0)),fl_range(fl(330),fl(330))),time(10,58,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(56),long_W(20)),fl_range(fl(330),fl(330))),
```

```
time(11,35,0)),0.8)belongs_to

profile_GAF033_1.

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(56),
long_W(20)),fl_range(fl(330),fl(330))),time(11,35,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(30)),fl_range(fl(330),fl(330))),
time(12,19,0)),0.8)belongs_to profile_GAF033_1.

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(30)),fl_range(fl(330),fl(330))),time(12,19,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(40)),fl_range(fl(330),fl(330))),
time(13,1,0)),0.8)belongs_to profile_GAF033_1.

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(40)),fl_range(fl(330),fl(330))),time(13,1,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(56),long_W(50)),fl_range(fl(330),fl(330))),
time(13,44,0)),0.8)belongs_to profile_GAF033_1.

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(56),
long_W(50)),fl_range(fl(330),fl(330))),time(13,44,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(54,37),long_W(55,52)),fl_range(fl(330),fl(330))),
time(14,12,0)),0.8)belongs_to profile_GAF033_1.

the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(54,37),
long_W(55,52)),fl_range(fl(330),fl(330))),time(14,12,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(53,37),long_W(58,8)),
fl_range(fl(330),fl(330))),time(14,25,0)),0.8)belongs_to profile_GAF033_1.
```

The testing strategy consisted of executing the existing executable CPS in order to discover possible conflict violation pairs from the profile data. Thus appropriate subsets were selected of the set of all possible pairs, viz. pairs of aircraft which were scheduled closely together. Most of the instances tested were in the category of *true negative* . However some pairs were eventually found in which conflict was detected. These constitute *false positive* instances. (N.B. However we do not have either *true positive* or *false negative* instances from this data.)

The following shows the result of a run of data from 30 false positive instances. Negated expressions are shielded, so that positive rules only are included. It is followed by a run of data from the same 30 false positive instances. Negated literals are expanded, so that both positive and negative rules are included. The query initiating blame assignment is as follows:

```
| ?-list_of_pos(List_of_instances),
   blame_assign_f_pos(List_of_instances,Blame_assignment).
```

where the predicate `list_of_pos` instantiates the list of (false) positive instances.

Run 1: Blame Assignment for 30 False Positive Instances – Negation Shielded

```
Blame_assignment =
[potential(1,30),potential(2,30),potential(3,30),potential(4,30),
potential(5,30),potential(6,30),potential(7,30),potential(8,28),
potential(9,2),potential(10,28),potential(11,28),potential(12,22),
potential(13,6),potential(14,9),potential(15,19),potential(16,30),
potential(18,9),potential(19,27),potential(20,28),potential(21,2),
potential(22,2),potential(23,2),potential(27,30),potential(29,9),
potential(35,30),potential(38,30),potential(47,30)]?
```

Run 2: Blame Assignment for 30 False Positive Instances – Negation Unshielded

```
Blame_assignment =
[potential(-51,30),potential(-50,30),potential(-48,20),potential(-46,30),
potential(-45,30),potential(-44,30),potential(-43,30),potential(-42,30),
potential(-41,30),potential(-39,30),potential(-32,30),potential(-31,30),
potential(-30,30),potential(-29,27),potential(1,30),potential(2,30),
potential(3,30),potential(4,30),potential(5,30),potential(6,30),
potential(7,30),potential(8,28),potential(9,2),potential(10,28),
potential(11,28),potential(12,22),potential(13,6),potential(14,9),
potential(15,19),potential(16,30),potential(18,9),potential(19,27),
potential(20,28),potential(21,2),potential(22,2),potential(23,2),
potential(27,30),potential(29,9),potential(35,30),potential(38,30),
potential(47,30),potential(48,10)]?
```

As can be seen, the information provided by Run 2 is greater than that provided by Run 1. Run 2 includes the additional information that rules with identities
-51, -50, -48, -46, -45, -44, -43, -42, -41, -39, -32, -31, -30, -29, 48
participate in the proofs of 'false positive' instances, and the number of instances where they occur. If a standard meta-interpreter were utilised this information would be hidden. The extended meta-interpreter has been applied to the CPS which has a large number of rules and resulting lengthy proof tree, and should be of great practical value.

The following is an example of a proof tree of one of the instances, both with and without the negative extension. The query which generates the trees is first presented, followed by proof trees 1 and 2. The Id's of the clauses only are given (because of space considerations). Tree 1 is that of a traditional meta-interpreter. Tree 2 utilises the extension to include negation explicitly. (Recall that '0' represents built-in predicates.)

Tree Generator – Negation Shielded

```
| ?- xgen_trees(
profiles_are_in_oceanic_conflict(the_Segment(profile_GAF033_1,
fourD_pt(threeD_pt(twoD_pt(lat_N(54,37),long_W(55,52)),fl_range(fl(330),fl(330))),
time(14,12,0)),fourD_pt(threeD_pt(twoD_pt(lat_N(53,37),long_W(58,8)),
fl_range(fl(330),fl(330))),time(14,25,0)),0.8),
     profile_GAF033_1,
     the_Segment(profile_DLH436_1,
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(40)),fl_range(fl(310),fl(310))),
time(13,24,0)),fourD_pt(threeD_pt(twoD_pt(lat_N(55,0),long_W(56,12)),
```

```
fl_range(fl(330),fl(330))),time(14,32,0)),0.84), profile_DLH436_1 ),
     profiles_are_in_oceanic_conflict(Seg1, Prof1,
Seg2, Prof2), _, _, GenIds).
```

Tree 1: Generated by one of the Instances – Negation Shielded

```
Seg1 = the_Segment(profile_GAF033_1,
fourD_pt(threeD_pt(twoD_pt(lat_N(54,37),long_W(55,52)),fl_range(fl(330),fl(330))),
time(14,12,0)),fourD_pt(threeD_pt(twoD_pt(lat_N(53,37),long_W(58,8)),
fl_range(fl(330),fl(330))),time(14,25,0)),0.8),

Seg2 =
the_Segment(profile_DLH436_1,fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(40)),
fl_range(fl(310),fl(310))),time(13,24,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(55,0),long_W(56,12)),
fl_range(fl(330),fl(330))),time(14,32,0)),0.84),

Prof1 = profile_GAF033_1,
Prof2 = profile_DLH436_1,
GenIds =
[1,[100036,100048,2,[7,[3,[4,[10071,10066,5,[10071,10066,10071,10078,
10067,10056,10033,0],6,[10071,10066,10071,10079,10068,10056,10034,0],10094,
10026,0,0],4,[10071,10066,5,[10071,10066,10071,10078,10067,10056,10033,0],6,
[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,0,0],10080,0],10076,
0],8,[10075,3,[4,[10071,10066,5,[10071,10066,10071,10078,10067,10056,10033,0],
6,[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,0,0],4,
[10071,10066,5,[10071,10066,10071,10078,10067,10056,10033,0],6,
[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,0,0],10080,0],0,12,
[10078,10113,0],14,[10079,10113,0],10,[1029,1029,10033,11,[1029,1033,1033,16,
[10135,10135,10135,10110,10135,10139,1034,10135,10135,10145,19,[0,27,
[10123,10123,1018,0],0],0,10135,10135,10145,20,[10075,35,[10071,0,10071,0,47,
[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,0,0,0,0,0,0,0,0],38,
[10071,10071,1022,0],0,0],10107,10122,0,0],0,0],16,[10135,10135,10135,10110,
10135,10139,1034,10135,10135,10145,19,[0,27,[10123,10123,1018,0],0],0,10135,
10135,10145,20,[10075,35,[10071,0,10071,0,47,[1012,10123,10123,1018,10123,
10123,1017,10071,10071,1024,0,0,0,0,0,0,0,0],38,[10071,10071,1022,0],0,0],
10107,10122,0,0],0,0],0],0],0],0],0]] ?
```

Tree 2: Generated by one of the Instances – Negation Unshielded

```
Seg1 =
the_Segment(profile_GAF033_1,fourD_pt(threeD_pt(twoD_pt(lat_N(54,37),
long_W(55,52)),fl_range(fl(330),fl(330))),time(14,12,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(53,37),long_W(58,8)),
fl_range(fl(330),fl(330))),time(14,25,0)),0.8),
Seg2 =
the_Segment(profile_DLH436_1,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(40)),fl_range(fl(310),fl(310))),time(13,24,0)),
```

```
fourD_pt(threeD_pt(twoD_pt(lat_N(55,0),long_W(56,12)),
fl_range(fl(330),fl(330))),time(14,32,0)),0.84),
Prof1 = profile_GAF033_1,
Prof2 = profile_DLH436_1,
GenIds =
[1,[100036,100048,2,[7,[3,[4,[10071,10066,5,[10071,10066,10071,10078,10067,
10056,10033,0],6,[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,
-101,0],4,[10071,10066,5,[10071,10066,10071,10078,10067,10056,10033,0],6,
[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,-101,0],10080,0],
10076,0],8,[10075,3,[4,[10071,10066,5,[10071,10066,10071,10078,10067,10056,
10033,0],6,[10071,10066,10071,10079,10068,10056,10034,0],10094,10026,-101,0],
4,[10071,10066,5,[10071,10066,10071,10078,10067,10056,10033,0],6,[10071,10066,
10071,10079,10068,10056,10034,0],10094,10026,-101,0],10080,0],-30,[[-31,
[[10084,10084,10088,10088,-10013]],-32,[[-10085]]]],12,[10078,10113,0],14,
[10079,10113,0],10,[1029,1029,10033,11,[1029,1033,1033,16,[10135,10135,10135,
10110,10135,10139,1034,10135,10135,10145,19,[-29,[[10092,10009,10091,10012,
10091,10012,0,-1032,10091,-10009,10091,-10009,10092,-10009]],27,[10123,
10123,1018,0],0],0,10135,10135,10145,20,[10075,35,[10071,-103,10071,-103,47,
[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-39,[[-1011]],-41,
[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-50,[[-1012]],-42,[[1012,10123,10123,1018,10123,
10123,1017,10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],-43,
[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-44,[[1012,10123,10123,1018,10123,10123,1017,
10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],-45,[[1012,10123,10123,
1018,10123,10123,1017,10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],-46,
[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-51,[[1012,10123,10123,1017,10071,10071,-1025]],0],
38,[10071,10071,1022,0],0,0],10107,10122,0,0],0,0],16,[10135,10135,10135,
10110,10135,10139,1034,10135,10135,10145,19,[-29,[[10092,10009,10091,10012,
10091,10012,0,-1032,10091,-10009,10091,-10009,10092,-10009]],27,[10123,10123,
1018,0],0],0,10135,10135,10145,20,[10075,35,[10071,-103,10071,-103,47,
[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-39,[[-1011]],-41,
[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-50,[[-1012]],-42,[[1012,10123,10123,1018,10123,
10123,1017,10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],-43,
[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-44,[[1012,10123,10123,1018,10123,10123,1017,
10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],-45,[[1012,10123,10123,
1018,10123,10123,1017,10071,10071,1024,-48,[[1012,10150,-1052,-1062]]]],
-46,[[1012,10123,10123,1018,10123,10123,1017,10071,10071,1024,-48,
[[1012,10150,-1052,-1062]]]],-51,[[1012,10123,10123,1017,10071,10071,-1025]],0],
38,[10071,10071,1022,0],0,0],10107,10122,0,0],0,0],0],0],0],0],0]]
?
```

Example:
The clause with Id '10009' denotes the infix 'is north of', e.g. 'latN(52) is north of latN(0)'.

An example of the clause with Id '-10009' is ' `not__(lat_N(57)is_north_of lat_N(58))`'.

In Figure 1 is a pictorial representation of a small fragment of Tree 1. In Figure 2, the same fragment is shown, with the negated literal expanded. In the second tree information as to the success of Rule -30 '`not_deemed_laterally_separated`' is provided by the expansion of this node.

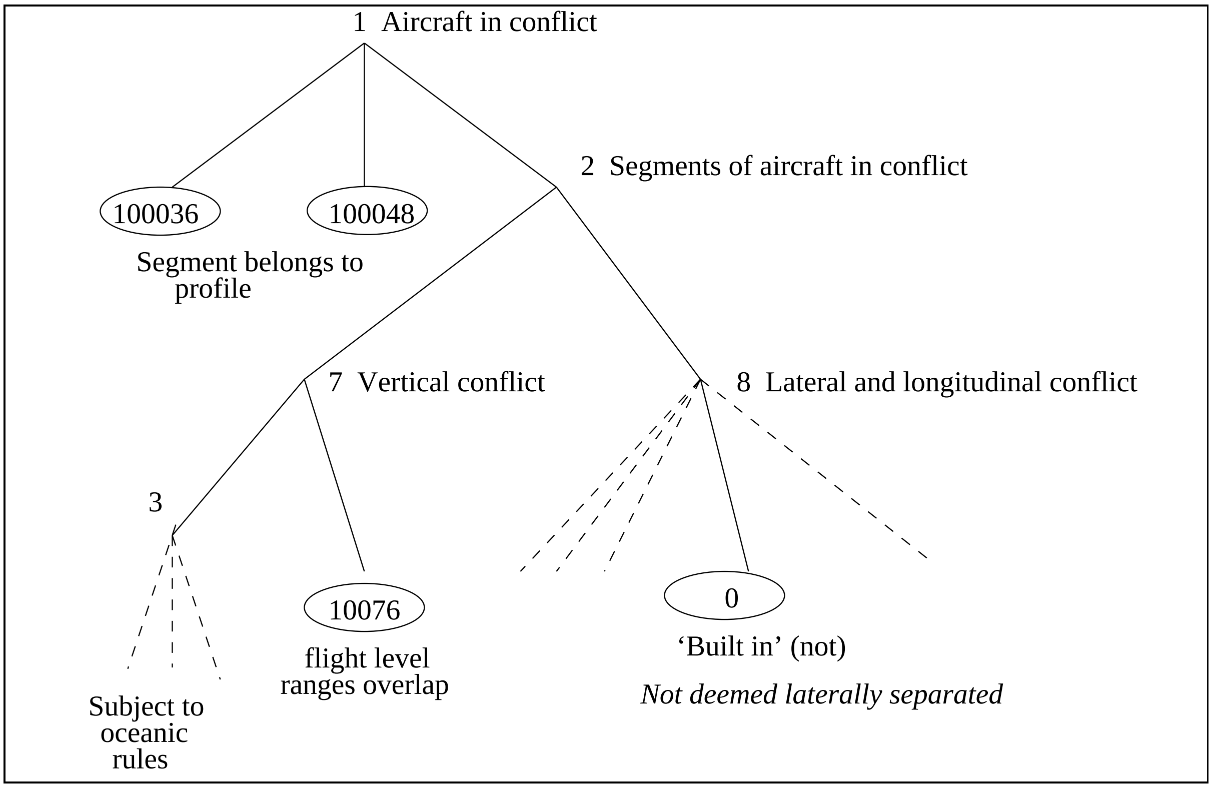

Figure 1: A Proof Tree Fragment of instance GAF033/ DLH436 - Negation shielded.

5 Analysis of Prolog Programs: Identifying Redundancy

One approach to debugging Prolog programs is to begin by executing queries. A query which fails when it was expected to succeed or succeeds when it was expected to fail alerts the programmer to the presence of an error. This debugging strategy is automatically performed as part of the method for assigning blame used by the IMPRESS theory revision program (see previous Sections of this report and [Bry97]). However this debugging strategy will not uncover all the errors in a program. A query that should succeed may do so but for the wrong reasons. Similarly a query that should fail may do so but for the wrong reasons. One drawback of the strategy is that it does not identify redundant clauses in a Prolog program. The following potential sources of redundancy are considered here.

1. Clauses which never fail (*NF*);
2. Clauses which never succeed (*NS*);

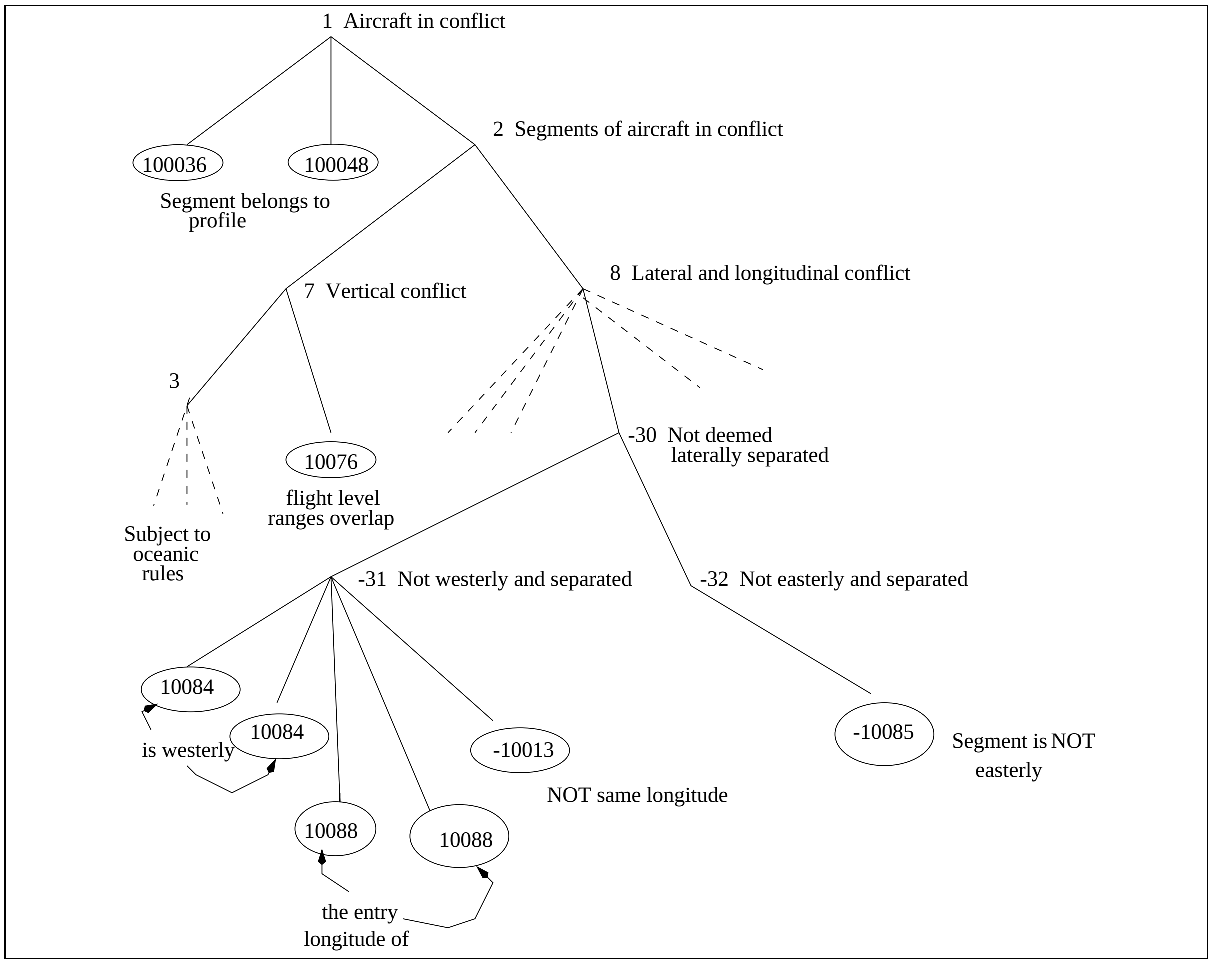

Figure 2: A Proof Tree Fragment of instance GAF033/ DLH436 – Negation expanded.

3. Clauses which are never reached (NR) .

Section 5.1 describes a method for attempting to identify these sources of redundancy using a set of test queries. In other words it explains how NF, NS and NR may be computed. Section 5.2 describes the algorithms for the method and their implementation. Note that a clause which is never reached is not necessarily unreachable. To discover unreachable clauses generally requires formal reasoning and this is not used here. Instead a statistical approach is taken which relies on the existence of a set of test queries which is representative of all the possible queries to the program. Thus the method identifies clauses which *apparently* perpetually succeed, clauses which *apparently* perpetually fail and clauses which *apparently* are never reached.

5.1 Computing Redundant Clauses

Let

C be the set of identifiers for the clauses in a Prolog program;
S_q be the set of identifiers for the clauses which succeed whilst executing a Prolog query, `q`;
F_q be the set of identifiers for the clauses which fail whilst executing `q`;
$Sproof_q$ be the set of identifiers for the clauses which succeed whilst executing `q` and which are part of the proof of that query.

It follows that
$Sproof_q \subset S_q$;
Reached code $= S_q \cup F_q$;
Unreached code $= C - (S_q \cup F_q)$.

We can extend these ideas with respect to a set of queries, Q, as follows. Let

X be the number of queries in Q;
S be the set of identifiers for the clauses which succeed whilst executing Q;
F be the set of identifiers for the clauses which fail whilst executing Q;
$Sproof$ be the set of identifiers for the clauses which succeed whilst executing Q and which are part of the proof of that query;
NF be the set of identifiers for the clauses which *never* fail whilst executing Q;
NS be the set of identifiers for the clauses which *never* succeed whilst executing Q;
NR be the set of identifiers for the clauses which are *never* reached whilst executing Q.

It follows that
$S = S_{q1} \cup S_{q2} \cup \ldots \cup S_{qX}$;
$F = F_{q1} \cup F_{q2} \cup \ldots \cup F_{qX}$;
$NF = C - F$;
$NS = C - S$;
$NR = C - (NF \cup NS)$.

NF, NS and NR are equivalent to items 1, 2 and 3 listed at the start of Section 5.

5.2 Implementation

It is clear that the computation of NF, NS and NR is trivial given S and F. The part of the theory revision program for blame assignment for False-Negatives (see [Bry97]) may be used to compute F. Note that the part for blame assignment for False-Positives may be used to compute *Sproof* but not S (see early Sections of this report). An additional algorithm is required for computing S. This is shown in Figure 3 and the corresponding code is listed in Appendix F.

```
1 For each instance
   form list of IDs for unshielded rules that succeed
   remove duplicates from this list
2 Append together all the uniqued ID-lists for the instances and sort
the new list into ascending order.
3 Remove duplicates from the sorted list, annotating it with counts.
```

Figure 3: Algorithm for Computing S

6 Conclusions

The 'text-book' meta-interpreters for generating proof trees are limited to definite programs. A meta-interpreter is needed which can generate proof trees which explicitly represent *negative* rules from general logic programs. As far as these authors are aware the work described in this report is a new contribution to the field. (It was presented at LOPSTR '97 [WBM97].) The extended meta-interpreter has been applied to the CPS which has a large number of rules and resulting lengthy proof tree, and should be of great practical value.

The output of the tree generator is such that it can be utilised by a 'blame assignment' algorithm – information about negated predicates is retained and not lost as in more traditional methods.

References

[AB94] K.R Apt and R.N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19(20):9–71, 1994.

[Bra90] I. Bratko. *Prolog. Programming for Artificial Intelligence*. Addison-Wesley, second edition, 1990.

[Bry97] C.H. Bryant. Deliverable 2.2: Theory revision. Technical Report impress/2/02/1, School of Computing and Mathematics, University of Huddersfield, UK, 1997.

[Cla78] K L Clark. Negation as Failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[dSP88] J.L. de Siqueira and J.F. Puget. Explanation-based generalisation of failures. In *Proc. ECAI-88*, pages 339–344, 1988.

[FT94] L. Fribourg and F. Turini, editors. *Proceedings of the Fourth International Workshops, LOPSTR '94 and META '94, Pisa, Italy, June 20-21, 1994, LNCS 883*, June 1994.

[KCM87] S.T. Kedar-Cabelli and L.T. McCarty. Explanation-based generalisation as theorem proving. In *Proceedings of the 4th International Workshop on Machine Learning, Irvine*, 1987.

[Llo87] J W Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second, extended edition edition, 1987.

[MPN+95] T.L. McCluskey, J.M. Porteous, Y. Naik, C.N. Taylor, and S. Jones. A requirements capture method and its use in an air traffic control application. *Software - Practice and Experience*, 25(1):47–71, 1995.

[Pet92] A. Pettorossi, editor. *Proceedings of the Third International Workshop, META-92 Uppsala, Sweden, June 10-12, 1992, LNCS 649*, June 1992.

[RM95] B. L. Richards and R. J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, May 1995.

[Sch95] S. Schrödl. An extension of explanation-based generalisation to negation as failure. In *Proceedings of the 19th Annual German Conference on Artificial Intelligence, Bielefeld, LNAI Vol. 981*, pages 65–76. Springer, 1995.

[Sch96] S. Schrödl. Explanation-based generalisation for negation as failure and multiple examples. In W. Wahlster, editor, *ECAI 96*, pages 448–452, Budapest, 1996. John Wiley & Sons.

[She96] J Shepherdson. Negation as failure, completion and stratification. In D M Gabbay, C J Hogger, and J A Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume V. Oxford University Press, UK, 1996. to appear.

[SS94] L Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, second edition, 1994.

[WB96] M.M. West and C.H. Bryant. Deliverable 2.1: Customisation of the formal requirements engineering environment for the IMPRESS project. Technical Report impress/1/02/1, School of Computing and Mathematics, University of Huddersfield, UK, 1996.

[WBM97] M.M. West, C.H. Bryant, and T.L. McCluskey. Transforming general program proofs: A meta interpreter which expands negative literals. In *LOPSTR '97*, Leuven, Belgium, 1997.

A Terminology

These appendices formalise the work presented in the main sections. We assume the standard logic program terminology. Thus a *general* clause (rule) has the form $H \longleftarrow \mathcal{B}$, with head H and body $\mathcal{B}$. $\mathcal{B}$ is composed of a conjunction of literals, L_i, which are expressed $L_1 \wedge L_2, \ldots, \wedge L_n$. Other denotations are $L_1, L_2, \ldots, L_n$, or $\bigwedge_i L_i$. In the latter case the limits will be understood as being from unity to some appropriate finite number. In a similar manner $L_1 \vee L_2 \vee, \ldots, \vee L_n$ can be denoted $\bigvee_i L_i$. Bold letters are used to denote finite sequences of syntactic objects, thus $x_1 = t_1, \ldots x_n = t_n$ is denoted $\mathbf{x} = \mathbf{t}$. Given that a substitution θ is a function from variables to terms, we write $E\theta$ for the result of applying θ to expression E. If $\mathcal{F}$ is a formula, then $\exists(\mathcal{F})$ denotes the existential closure of $\mathcal{F}$, where all its free variables are existentially quantified. In a similar manner, $\forall(\mathcal{F})$ denotes the universal closure of $\mathcal{F}$.

B Negation as Failure and Completion

SLD-resolution allows the derivation of positive consequences (namely, conjunctions of atoms) [AB94] from Horn clause programs. Where negative consequences are desired, in general programs, SLD-resolution is augmented with the *Negation as Failure rule* to become *SLDNF-resolution*. (See also [Llo87, She96].) In order to justify the use of negation as failure rule, Clark [Cla78] introduced the idea of the completion of a general program, *Prog*, and this is outlined as follows.

The *completed* definition of predicate p ($\in$ *Prog*) requires a new predicate '=' whose intended interpretation is identity. Suppose predicate p is defined by m statements of the form: $p(\mathbf{t_i}) \longleftarrow W_i$, where W_i is a conjunction of literals. The completed definition of p is a disjunction:

$$\boxed{\forall \mathbf{x}(\ p(\mathbf{x}) \longleftrightarrow \bigvee_i \exists \mathbf{y_i}(\mathbf{x} = \mathbf{t_i}) \ \wedge \ W_i\ ) \qquad \textbf{(N1)}}$$

where $\mathbf{y_i}$ are the variables of the original clause. Additionally, if q is a proposition or predicate occurring in a program, where there is *no* program statement with q at its head, the completed definition of q is $\forall \mathbf{x} \neg\ q(\mathbf{x})$. ($q$ is 'undefined'.)

For SLDNF resolution, positive literals are 'deleted' via resolution. The proposed solution [AB94] for negative literals is (intuitively) as follows: the deletion of every negative literal is via a subsidiary (finitely failed) tree. A proof tree for a query containing negative literals is composed of a 'main' tree and subsidiary trees associated with negative literals. The subsidiary trees are 'kept aside' from the main tree. For each node n associated with a negative literal, a subsidiary tree is functionally linked to the main tree.

C Tree Generation – Definitions

C.1 Tree Definition – Negation Shielded

First we consider the case where *negation* is shielded. The definition presented here is based on 'traditional' EBG tree generation described in [KCM87]. A recursive function *gen_tree* takes a non-empty goal G and a node n and yields an expression as follows. Our development and notation follows that of [Sch95], in order for later comparison. The tree generation is guided by an instance α whose role is to decide which clauses are used in a resolution step. It produces a *generalised* version of the proof of the instance which follows the proof. (The example in Section 3.2 shows both proofs.) The tree generation is assumed independent of the computation rule. Consider root node n of SLD tree labelled with instance $G\alpha$ of G. Suppose G has the form

$$\mathcal{L}, p(\mathbf{t}), \mathcal{R},$$

where $\mathcal{L}, \mathcal{R}$ are sets of conjoined literals left and right of $p(\mathbf{t})$. Suppose non-empty G. Assuming $p(\mathbf{t})\alpha$ is the atom selected at n then there are two cases to consider: predicate $p(\mathbf{t})$ can be shielded or unshielded. If $p(\mathbf{t})$ is shielded, then it is eliminated via resolution using other shielded predicates and *gen_tree* calls itself recursively with goal argument $(\mathcal{L}, \mathcal{R})$ and node m. If predicate $p(\mathbf{t})$ is unshielded, suppose node m is a child of n on a successful branch derived with clause $p(\mathbf{s})$ where $p(\mathbf{s}) \longleftarrow \mathcal{B}$. Node m is labelled $(\mathcal{L}, \mathcal{B}, \mathcal{R})\alpha\theta$ where $\mathbf{t}\alpha\theta = \mathbf{s}\alpha\theta$. The clause $p(\mathbf{s}) \longleftarrow \mathcal{B}$ eliminates $p(\mathbf{t})$ and *gen_tree* calls itself recursively with goal argument $(\mathcal{L}, \mathcal{B}, \mathcal{R})$ and node m. The tree $\mathit{gen\_tree}(G, n)$ is defined as:

$$\begin{aligned} \mathit{gen\_tree}(G, n) &= G : G = \Box && (1)\\ &= p(\mathbf{t}), \mathit{gen\_tree}((\mathcal{L}, \mathcal{R}), m) : p \text{ is shielded;} && (2)\\ &= (\mathbf{t} = \mathbf{s}), \mathit{gen\_tree}((\mathcal{L}, \mathcal{B}, \mathcal{R}), m) : p \text{ is unshielded} && (3) \end{aligned}$$

The equality $(\mathbf{t} = \mathbf{s})$ in (3) represents the instantiation of the new goal $(\mathcal{L}, \mathcal{B}, \mathcal{R})$. Note that (2) includes the case where the 'atom' considered at n is a negated literal.

C.2 Tree Definition – Negation Expanded

For the most part our second definition of proof tree expansion is the same as the first, apart from the treatment of the case where the 'atom' considered at n is a negated literal, previously regarded as shielded.

In the extended tree, goals can take the form $(\mathcal{L}, (\mathcal{E}), \mathcal{R})$, as well as $(\mathcal{L}, p(\mathbf{t}), \mathcal{R})$, where $\mathcal{E}$ is a negated conjunction of literals, $\neg \bigwedge_j L_j$. We first suppose a goal $(\mathcal{L}, \neg\, q(\mathbf{t}), \mathcal{R})$. We consider *all* clause heads matching $q(\mathbf{t})$. Suppose $q(\mathbf{t})$ fails. We are assuming that *either* $q(\mathbf{t})$ is ground *or* all non-ground variables are existentially quantified, $\neg\, \exists\, q(\mathbf{t})$. If q is unshielded then recall that each of the matching clause heads is from an unshielded predicate. Thus node m is a child of n on a successful branch derived with clause $\neg\, q(\mathbf{t})$. From **N1** (the completed definition of q):

$$\neg\, \exists\, q(\mathbf{t}) \longleftrightarrow \neg\, \exists(\bigvee_i \exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ W_i) \longleftrightarrow \forall(\bigwedge_i \neg\, (\exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ W_i.))$$

The above unfolding process corresponds to (5) below and is the first step in the expansion of the negative tree. Recall that the process stops only when a component predicate is *shielded*. No variables are instantiated in $\mathcal{L}, \mathcal{R}$, for these are outside the scope of the existential quantifiers. Given an input goal $(\mathcal{L}, \neg\, q(\mathbf{t}), \mathcal{R})$, *gen_tree* eliminates $\neg\, q(\mathbf{t})$ and *gen_tree* calls itself recursively for each goal argument $\neg\, (\exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \;\; \wedge \;\; W_i)$. (The equalities in the expression correspond to matches of $q(\mathbf{t})$.) The resultant expressions are then conjoined, for it is necessary for each of the conjoined components, comprising the definition of $q(\mathbf{t})$ to fail for $q(\mathbf{t})$ to fail.

The second stage of the process of dealing with negative literals corresponds to (6) below and consists of the transformation of each clause body $(\neg\, \exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \;\; \wedge \;\; W_i)$, as follows. Suppose the clause body W_i is a conjunction of literals $\bigwedge_j L_{ij}$. The variables in the set $\{L_{ij}\}$ may be shared. (An example is `m(A,X)` in Section 3.2.) In order to capture the contribution of all the literals, we include the subset of literals which share variables and which succeed. Lemma 1 and its corollary justifies the step.

Lemma 1: Expansion of Negated term:
$\neg\, (\exists\, \mathbf{y} \;\; \bigwedge_{j=1}^{n} E_j) \longleftarrow \forall\, \mathbf{z}(\;\; ((\forall\, \mathbf{y_A} \;\; \bigwedge_{\alpha\in A} E_\alpha) \wedge (\bigvee_{\beta\in B} \neg\, \exists\, \mathbf{y_B} E_\beta)) \vee (\neg\, \exists\, \mathbf{y_A} \;\; \bigwedge_{\alpha\in A} E_\alpha) \;\;)$
where

1. A and B arbitrarily partition $\{1, \ldots, n\}$,
2. $\mathbf{z}$ are the variables shared by $\{E_\alpha\}, \{E_\beta\}$,
3. $\mathbf{y_A}$ are the variables in $\{E_\alpha\}$ but not in $\{E_\beta\}$,
4. $\mathbf{y_B}$ are the variables in $\{E_\beta\}$ but not in $\{E_\alpha\}$,

(Thus $\mathbf{z}, \mathbf{y_A}, \mathbf{y_B}$ partition the variables of $\mathbf{y}$.)
Proof:

$$
\begin{aligned}
\neg\, (\exists\, \mathbf{y} \;\; \textstyle\bigwedge_{j=1}^{n} E_j) &\longleftrightarrow \neg\, \exists\, \mathbf{y} \; (\textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \bigwedge_{\beta\in B} E_\beta) && [\text{Definition of } A, B] \\
&\longleftrightarrow \forall\, \mathbf{y} \neg\, (\; \textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \bigwedge_{\beta\in B} E_\beta) && [\text{Quantifier property}] \\
&\longleftrightarrow \forall\, \mathbf{y}(\;\; \neg\, (\textstyle\bigwedge_{\alpha\in A} E_\alpha) \vee \neg\, (\bigwedge_{\beta\in B} E_\beta) \;\;) \\
& && [\text{De Morgan}] \\
&\longleftrightarrow \forall\, \mathbf{y}((\; \textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \neg \; \bigwedge_{\beta\in B} E_\beta) \vee \neg \;\; \bigwedge_{\alpha\in A} E_\alpha) \\
& && [\text{From the equivalence } \neg\, A \vee B \leftrightarrow (A \wedge B) \vee \neg\, A] \\
&\longleftrightarrow \forall\, \mathbf{z} \forall\, \mathbf{y_A} \forall\, \mathbf{y_B} \; ((\; \textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \neg \; \bigwedge_{\beta\in B} E_\beta) \vee \neg \;\; \bigwedge_{\alpha\in A} E_\alpha) \\
& && [\text{Partition of } \mathbf{y}] \\
&\longleftarrow \forall\, \mathbf{z}(\;\; \forall\, \mathbf{y_A} \forall\, \mathbf{y_B}(\; \textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \neg \; \bigwedge_{\beta\in B} E_\beta) \vee (\forall\, \mathbf{y_A} \forall\, \mathbf{y_B}(\neg \;\; \bigwedge_{\alpha\in A} E_\alpha)) \;\;) \\
& && [\text{Distribution of } \forall \text{ over } \vee \,] \\
&\longleftarrow \forall\, \mathbf{z}(\;\; (\, (\forall\, \mathbf{y_A} \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge (\forall\, \mathbf{y_B} \neg \; \bigwedge_{\beta\in B} E_\beta) \,) \vee (\neg\, \exists\, \mathbf{y_A} \;\; \bigwedge_{\alpha\in A} E_\alpha) \;\;) \\
& && [\text{Definition of } \mathbf{y_A}, \mathbf{y_B}, \text{ Distribution of } \forall \text{ over } \wedge \,] \\
&\longleftarrow \forall\, \mathbf{z}(\;\; (\, (\forall\, \mathbf{y_A} \;\; \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge (\forall\, \mathbf{y_B} \bigvee_{\beta\in B} \neg\, E_\beta) \,) \vee (\neg\, \exists\, \mathbf{y_A} \;\; \bigwedge_{\alpha\in A} E_\alpha) \;\;) \\
& && [\text{De Morgan}] \\
&\longleftarrow \forall\, \mathbf{z}(\;\; (\, (\forall\, \mathbf{y_A} \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge \bigvee_{\beta\in B} \forall\, \mathbf{y_B} \neg\, E_\beta \,) \vee (\neg\, \exists\, \mathbf{y_A} \;\; \bigwedge_{\alpha\in A} E_\alpha) \;\;) \\
& && [\text{Distribution of } \forall \text{ over } \vee \text{ -- implied by}]
\end{aligned}
$$

$$\longleftarrow \forall \mathbf{z}(\ ((\forall \mathbf{y_A} \bigwedge_{\alpha\in A} E_\alpha) \wedge \bigvee_{\beta\in B} \neg \ \exists \mathbf{y_B} E_\beta \ ) \vee (\neg \ \exists \mathbf{y_A} \ \bigwedge_{\alpha\in A} E_\alpha))$$

□

As a corollary:

$$\begin{aligned}&\neg \ \exists(\mathbf{y}_i(\mathbf{t} = t_i) \wedge \textstyle\bigwedge_{j=1}^{n} E_j) \longleftarrow \\ &\quad \forall \mathbf{z}_i(\ ((\forall \mathbf{y}_{iA}(\mathbf{t} = t_i) \wedge \ \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge (\bigvee_{\beta\in B} \neg \ \exists \mathbf{y}_{iB}(\mathbf{t} = t_i) \wedge E_\beta)) \\ &\qquad \vee (\neg \ \exists \mathbf{y}_{iA}(\mathbf{t} = t_i) \wedge \textstyle\bigwedge_{\alpha\in A} E_\alpha) \ \ )\end{aligned}$$

(Note that the converse is not necessarily true.) For the arbitrary subset $\{L_{i\alpha} : \alpha \in A\}$, we include only $L_{i\alpha}$ which succeed. Thus *failed* L_{ij} belong to the set $\{L_{i\beta} : \beta \in B\}$. It is sufficient for *one* of the $L_{i\beta}$ to fail for W_i to fail, and there may be more than one sub-tree associated with the failure of each W_i[2]. We thus consider one of the sub-trees and we suppose this to be the one associated with L_{ik}; we assume that L_{ik} fails. Then with input goal $\mathcal{L}, (\neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ W_i), \mathcal{R}$, the result is a further recursive call with new goal $\mathcal{L}, (\neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ L_{ik}), \mathcal{R}$. This then forms the input to either (5) or (7). (7) comprises the case where one or more of the disjoined literals is itself negative. Thus suppose that L_{ik} is of the form $\neg \ M$, then the goal becomes $\mathcal{L}, M, \mathcal{R}$.

Recalling that goals G can take the form $(\mathcal{L}, \neg \ (\mathcal{E}), \mathcal{R})$, $gen\_tree(G, n)$ is extended as:

$$\begin{aligned} gen\_tree(G,n) \ &= \ \neg \ q(\mathbf{t}), gen\_tree((\mathcal{L},\mathcal{R}), m) : \mathcal{E} \text{ is } \neg \ q(\mathbf{t}) \text{ and } q(\mathbf{t}) \text{ is shielded;} && (4)\\ &= \ \bigwedge_i gen\_tree((\mathcal{L}, (\neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ W_i), \mathcal{R}), m_i) : \mathcal{E} \text{ is } \neg \ \exists \, q(\mathbf{t}); && (5)\\ &= \ gen\_tree((\mathcal{L}, (\neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ L_{ik}), \mathcal{R}), m) \ \wedge \\ &\qquad \bigwedge_{\alpha\in A} gen\_tree((\mathcal{L}, ((\mathbf{t} = \mathbf{t_i}) \ \wedge \ L_{i\alpha}), \mathcal{R}), m_\alpha) : \\ &\qquad \mathcal{E} \text{ is } \neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ \bigwedge_j L_{ij} \text{ and } \neg \ \exists \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) \ \wedge \ L_{ik} \text{ succeeds and} \\ &\qquad \forall \, \alpha \in A, L_{i\alpha} \text{ succeeds and shares variables with } L_{ik}; && (6)\\ &= \ gen\_tree(\mathcal{L}, M, \mathcal{R}) : \mathcal{E} \text{ is } \neg \ (\neg \ M). && (7)\end{aligned}$$

D Previous Work

Siqueira and Puget have described a method for generating a failed proof tree ([dSP88]), namely, *Explanation-Based Generalisation of Failures (EBGF)*. A sufficient condition is derived from the failed proof tree which is satisfied by the instance and ensures the failure of the goal. However clause bodies contributing to the failed tree can contain only positive literals. The work has subsequently been extended [Sch96, Sch95]: EBGF has been used to aid the generation of trees for proofs which use SLDNF-resolution. General clauses can be associated with the tree. ([Sch95] also includes a review of other methods.)

[2]This limitation, of not being able to capture all the proofs in one tree is not confined to negative trees.

D.1 EBGF - method

The method uses the definition of program completion (Section B) as follows:

Given: A goal, G (a counterexample resulting in a failed proof tree).

Completed Definition and Unfolding: Each predicate p can be defined as a disjunction:
$\forall x_1 \ldots \forall x_n (p(x_1, x_2, \ldots, x_n) \longleftrightarrow A_1 \vee \ldots A_m)$. Starting with G, we unfold each conjoined component,A_i. Recall that each A_i is a conjunction of literals, we replace each literal with its completed definition. The rewriting is completed when all the derived predicates are 'operational'.

Simplification: The distributivity of 'or' over 'and' is applied to put the result into disjunctive form. Negating the result gives a a conjunction of negated components, B_i, where each B_i is itself a conjunction of literals: $\forall x_1 \ldots \forall x_n (p(x_1, x_2, \ldots, x_n) \longleftrightarrow \neg B_1 \wedge \ldots \wedge \neg B_m)$.

Removal of Literals: The resulting generalisation may be very complex so a heuristic is used to remove literals from each of the B_i. Sufficient literals are retained to obtain a condition satisfied by the counterexample.

D.2 EBGF - extension

The method is extended by Schrödel in [Sch96, Sch95], using traditional EBG described in [KCM87]. For positive literals, a traditional EBG tree is generated. However for negative literals a subsidiary tree is generated via EBGF. The *ebg* tree is defined in a similar manner to *gen_tree* described by equations (1-3). However a subsidiary *ebgf* tree is also defined as follows. If n is a node of a *failed* SLD tree with instance $G\alpha$, where the set of $p(\mathbf{t_i} \longleftarrow B_i$ defines p, the children of n are the set of n_i. Then

$$\begin{aligned} ebgf(G,n) &= p(\mathbf{t}), \bigwedge_i ebgf(((\mathcal{L},\mathcal{R}),n_i) : p(\mathbf{t}) \text{ is shielded;} && (8)\\ &= \bigvee_i (\mathbf{t} = \mathbf{t_i}) ebgf((\mathcal{L}, B_i, \mathcal{L}), n_i) : p(\mathbf{t}) \text{ is unshielded} && (9) \end{aligned}$$

The *ebgf* tree is joined to the main tree via the function $subs(n)$ defined in Section B. The generator recurses between EBG and EBGF. The derived formula contain negative goals, disjunctions and existential quantifiers. It is then converted to a set of general clauses via translation rules provided in [Llo87].

The difference between the method described and our work is that the *ebgf* tree is defined separately from the *ebg* tree. In our work the failed clauses are redefined and integrated with the successful clauses. Thus negation is 'deferred' to the leaf nodes of the tree. This has the advantage that the failed clauses of interest, viz. the unshielded clauses are immediately identifiable.

The CPS has a large number of rules and resulting lengthy proof tree, and thus we feel that our method is an improvement over *ebg*/*ebgf* just described.

E Blame Assignment Program Code

```
/* altered by mmw to expand out negated terms */
/*
 Code to generate a generalised proof tree - the code under analysis
can include negated literals

NB clause_impress/4 facts are assumed to exist prior to execution of
the code in this file.

*/

/* further alteration - obtain numerical tree - 20/4/97  MMW */

/* Predicate for obtaining numerical trees is called xgen_trees */

/*Code has been re-organised by mmw - August 1997 */

/*****************************************************************/
/*
Code for assigning blame to rules which succeed during an attempt to
prove a negative instance i.e. blame assignment code for FALSE
POSITIVES.

Algorithm
---

1 For each instance

        generate a generalised proof tree
        form list of unshielded rules which does not include facts
        form list of IDs for these rules
        remove duplicates from this list

2 Append together all the uniqued ID-lists for the instances and sort
the resulting list into ascending order.

3 Remove duplicates from the appended list, annotating it with counts.

NB Duplicate rules are identified using the ID.

NB clause_impress/4 facts are assumed to exist prior to execution of
the code in this file.

*/


/*****************************************************************/

/*
        blame_assign_f_pos(List_of_instances,Blame_assignment)

Currently, this is the top level blame assignment procedure.

*/
```

```
blame_assign_f_pos(List_of_instances,Blame_assignment):-

        f_pos_get_rules(List_of_instances,L),
    qsort(L, Sorted_L),
        rule_counter(Sorted_L,Blame_assignment).

/******************************************************************/



f_pos_get_rules([],[]).
f_pos_get_rules([H|T], Rules):-
        get_list_of_rules_used_in_proof(H, Hrules),
        tell(user), nl,
        write(Hrules), nl, told,
        f_pos_get_rules(T, Trules),
        append(Hrules, Trules, Rules).

/******************************************************************/



/* nb extra argument in xgen_trees
  gen_trees/4
The procedure gen_trees(X,Gen_X,Proof,Gen_proof) generates a proof
tree and a generalised proof tree. Both trees include clause IDs.

xgen_trees/5
The procedure xgen_trees(X,Gen_X,Proof,Gen_proof, GenIds) generates a proof
tree,  generalised proof tree and a tree of Ids.
 */


get_list_of_rules_used_in_proof(Instance,Unique_list_of_rule_IDs):-

        generalise_instance(Instance,Gen_Instance),
        tell(user), nl,
        write(Instance), nl, told,
        gen_trees(Instance,Gen_Instance,_,Gen_proof),
        form_rules(Gen_proof,L),
        form_list_of_rule_IDs(L,List_of_rule_IDs),
        remove_dups(List_of_rule_IDs,Unique_list_of_rule_IDs).

/******************************************************************/

/* The purpose of this rule is to generate the generalised version of
an instance needed for the call to gen_trees.
*/

generalise_instance(X,GenX):-

        functor(X,F,N),
        functor(GenX,F,N).

/******************************************************************/

/* The procedure form_rules(P,L) inputs a proof tree,P, and outputs a
```

```
list,L, of lists, where each sublist represents a rule used in the
proof. The sublists for rules take the form

        [Rule_ID, Head_of_rule, Subgoal_1, Subgoal_2.... ]

NB Facts used in the proof are not included in the L.

Here is some test data.

form_rules([1,[a,[2,a1]]], [[1,a,a1]]).
form_rules([5,[b,[6,[b1,[7,b1i]]]]], [[5,b,b1], [6,b1,b1i]]).
form_rules([8,[c,[9,c1]],10,[d,[11,d1]]],
          [[8,c,c1],[10,d,d1]]).
form_rules([12,e], []).
form_rules([12,e,13,f], [] ).
form_rules([10,[d,[11,d1]],12,e,13,f],
          [[10,d,d1]]).
form_rules([14,[h,[12,e,13,f]]], [[14,h,e,f]]).
form_rules([15,[i,[12,e,13,f,10,[d,[11,d1]]]]],
          [[15,i,e,f,d],[10,d,d1]] ).
form_rules([17,[k,[12,e,10,[d,[11,d1]],13,f]]],
          [[17, k, e, d, f],[10,d,d1]] ).
*/

form_rules([], []).

form_rules([N, X], []):-integer(N),   % facts are not returned
                        \+ list(X).

form_rules([N, [H, L] | T], Rules):-

        integer(N),
        \+ list(H),
        list(L),
        get_subgoals(L, Lsubgoals),
        form_rules(L, Lrules),
        form_rules(T, Trules),
        append([[N, H | Lsubgoals]], Lrules, Rules1),
        append(Rules1, Trules, Rules).


form_rules([H|T], Rules):-

        list(H),
        form_rules(H, Hrules),
        form_rules(T, Trules),
        append(Hrules, Trules, Rules).

form_rules([N, H|T], Rules):-

        integer(N),
        \+ list(H),
        form_rules([N, H], Hrules),
        form_rules(T, Trules),
        append(Hrules, Trules, Rules).

/*****************************************************************/
```

```
/* test with

get_subgoals([2,a1], [a1]).
get_subgoals([6,[b1,[7,b1i]]], [b1]).
get_subgoals([12,e,13,f], [e, f]).
get_subgoals([12,e,13,f,10,[d,[11,d1]]], [e, f, d]).

*/

get_subgoals([], []).

get_subgoals([N, A], [A]):-          integer(N),
                                  \+ list(A).

get_subgoals([N, [X|_] | T], [X | TSGs]):-

        integer(N),
        get_subgoals(T, TSGs).

get_subgoals([N, H | T], [H | TSGs]) :-          integer(N),
                                          \+ list(H),
                                          get_subgoals(T, TSGs).

/******************************************************************/

/* gen_trees/4
The procedure
gen_trees( +Expr, :Gen_Expr, ?P, ?GenP).

generates a proof tree and a generalised proof tree. Both trees
include clause IDs.

 Expr can take the form:

Case 1  Atom+: a positive literal;

Case 2 (Expr, Expr+): conjunction;

Case 3 (Expr ; Expr+): disjunction;

Cases (4a - 6)   not(Expr): negated expression.


xgen_trees/5
The procedure xgen_trees( +Expr, :Gen_Expr, ?P, ?GenP, ?GenIds)
generates a proof tree,  generalised proof tree and a tree of Ids.

For example:

rich2(A) :- not__(poor(A)).
month_sal(steve, 3000).

poor(A, 1) :- month_sal(A, X), Y is 12*X, Y < 20000 .
poor(A, 2) :- month_sal(A, X), Y is 12*X, Y < 24000 .
poor(A, 3) :- month_sal(A, X), Y is 12*X, Y < 36000 .
```

```
Given query:
-? gen_trees(rich2(steve), rich2(X), P, GenP).

unexpanded negation:

P
= [27,[rich2(steve),[-(28),[not__(poor(steve,2)),[0,not__(36000<24000)]],-(34),[not__(poor(steve,3)),
  [0,not__(36000<36000)]],-(35),[not__(poor(steve,1)),[0,not__(36000<20000)]]]]],
GenP = [27,[rich2(X),[-(28),[not__(poor(X,_A)),[0,not__(_B<24000)]],-(34),[not__(poor(X,_A)),
  [0,not__(_C<36000)]],-(35),[not__(poor(X,_A)),[0,not__(_D<20000)]]]]] ?

expanded negation:
result from new code - extra tree:

P = [27,[rich2(steve),[-28,[not__(poor(steve,2)),[0,not__(36000<24000)]],-34,[not__(poor(steve,3)),
  [0,not__(36000<36000)]],-35,[not__(poor(steve,1)),[0,not__(36000<20000)]]]]],
GenP = [27,[rich2(X),[-28,[not__(poor(X,2)),[0,not__(_A<24000)]],-34,[not__(poor(X,3)),
[0,not__(_B<36000)]],-35,[not__(poor(X,1)),[0,not__(_C<20000)]]]]] ?

*/

/*************************

Converts predicate with 5 arguments to the usual EBG predicate - 4
arguments.

****************************************/
gen_trees(Expr, Gen_Expr, P, GenP) :- xgen_trees(Expr, Gen_Expr, P, GenP, _).


/*******************************************
Cases 1- 4 are examined first:
******

Case 1. Expr a positive literal:
Processing depends on whether Expr matches the head of a shielded, or
unshielded predicate.

*****************************************************************/


xgen_trees(Head, Gen_head, [Clause_ID_no, Proof], [Clause_ID_no,
Gen_proof],  GenIds ) :-

     \+(Head = not__(_)),\+(Head = ground_not__(_)),
    clause_impress(Gen_head, Gen_body, Clause_ID_no, Shield_status),
    copy( (Gen_head:-Gen_body), (Head:-Body) ),
    Body,
              %% Body is called to check that it succeeds.
              %% This is necessary because Head may match the
              %% head of a Prolog procedure rather than just
              %% a single rule. If Body fails then Prolog will
              %% backtrack to the other rules in the procedure.
    !,
    ((Shield_status = shielded,  GenIds =  [Clause_ID_no],
       Proof = Head, Gen_proof =  Gen_head)
     ;
```

```
        (Shield_status = unshielded,  GenIds =  [Clause_ID_no, Body_Ids],
         process_body(Shield_status, Head, Gen_head, Body, Gen_body,
                              Proof, Gen_proof,   Body_Ids))).

/****************************************************************/


process_body(unshielded, Head, Gen_head, Body, Gen_body,
                      Proof, Gen_proof, GenIds):-
        tell(user),   nl,
        write('Case 1 - processing non negative body'), nl,
        told,
        xgen_trees(Body, Gen_body, Body_proof, Gen_body_proof,  GenIds),
        append([Head],[Body_proof], Proof),
        append([Gen_head], [Gen_body_proof], Gen_proof).

/* Head is built-in */
/* built-in predicates are given an ID of 0.*/


xgen_trees(A, Gen_A, [0, A], [0, Gen_A], [0]) :-
        built_in(A),
        tell(user),   nl,
        write('Case 1 - processing built-in'), nl,
        told,
        !,
        A.

/* Head matches with a fact */

xgen_trees(Head,Gen_head,[Clause_ID_no, Head],[Clause_ID_no,
Gen_head], [Clause_ID_no]):-
          \+(Head = not__(_)),  \+(Head = ground_not__(_)),
        clause_impress(Head,true,Clause_ID_no,_),
        tell(user),   nl,
        write('Case 1 - processing fact'), nl,
        told,
        !.

/* Case 2 - conjunction of two expressions */

xgen_trees((A, B), (Gen_A, Gen_B), Proof, Gen_proof,GenIds) :-
        !,
        tell(user),   nl,
        write('Case 2 - processing conjunction'), nl,
        told,
        xgen_trees(A, Gen_A, A_Proof, Gen_A_Proof, Gen_A_Ids),
        xgen_trees(B, Gen_B, B_Proof, Gen_B_Proof, Gen_B_Ids),
        append(A_Proof, B_Proof, Proof),
        append(Gen_A_Proof, Gen_B_Proof, Gen_proof),
              append(Gen_A_Ids, Gen_B_Ids, GenIds) .

/* Case 3 - disjunction of two expressions using ' ; ' */

xgen_trees((A ; _), (Gen_A ; _), Proof, Gen_proof, GenIds) :-
        tell(user),   nl,
        write('Case 3 - processing disjunction'), nl,
```

```
        told,
        xgen_trees(A, Gen_A, Proof, Gen_proof, GenIds),
        !.

xgen_trees((_ ; A), (_ ; Gen_A), Proof, Gen_proof, GenIds) :-
        !,
        xgen_trees(A, Gen_A, Proof, Gen_proof,GenIds).

/* for traditional EBG 'not(Expr)' is treated as a
'built-in' predicate, viz. Case 4
*/

/************
Cases 4a -6. not(Expr)
*/

/* case 4a not(Literal)

unfolds queries of the type not__(A) where A :- B into
not__(B). All  clauses A which match with Literal are
considerered. Thus the code generates a set of clause ids whose head
has the same predicate name as the input clause.
*/

xgen_trees(not__(A), not__(Gen_A) , Set_of_Heads, Set_of_GenHeads, GenIds) :-

   \+(A = not__(_)),

     tell(user),     nl,
      write('unfolding and collection of head matches - case 4a'),
      nl,      told,
      copy_term(Gen_A, Copy_Gen_A),

/* see below for examples of this use of setof */

      setof(N,
       G_B^ clause_impress( Copy_Gen_A, G_B, N , unshielded) ,
                               Set_of_Ids),

      tell(user),     nl,
      write( Set_of_Ids),
      nl,      told,
      gen_conjoined_set( not__(A), Gen_A, Set_of_Ids, Set_of_Heads,
                          Set_of_GenHeads, GenIds).

/* example: setof There are 2 options, the first of which is the most
general.


rule 28 poor(A, 24000) :- month_sal(A, X), Y is 12*X, Y < 24000 .
rule 34 poor(A, 36000) :- month_sal(A, X), Y is 12*X, Y < 36000

Query:
setof(N, X^(A^(B^(clause_impress(poor(A,X),B,N, unshielded)))), S).

gives answer
```

```
S = [28,34] ? ;


rule 28 poor(A, 24000) :- month_sal(A, X), Y is 12*X, Y < 24000 .
rule 34 poor(A, 36000) :- month_sal(A, X), Y is 12*X, Y < 36000

Query:
setof(N, X^(A^(B^(clause_impress(poor(A,X),B,N, unshielded)))), S).

gives answer

S = [28,34] ? ;

*/

/* base case -generates conjoined set of proofs */

gen_conjoined_set(_, _, [], [], [], []).

gen_conjoined_set(not__(A), Gen_A, [Id  | Rest],
                 [NegId,[not__(Copy_A), Proof]| RestProofs],
                 [NegId, [not__(Copy_Gen_A), Gen_proof]|RestGenProofs],
                 [NegId, [GenIds]|RestGenIds]) :-

         NegId is -Id,

         copy_term(A, Copy_A),
         copy_term(Gen_A, Copy_Gen_A),
         clause_impress(Copy_Gen_A, _, Id, unshielded),
      %%  subsumes(Copy_Gen_A,Gen_A),
      %%need to unify the above without changing Gen_A
         rename(Gen_A, Copy_Gen_A),
         clause_impress(Copy_Gen_A, Gen_B, Id, unshielded),
        copy((Copy_Gen_A:-Gen_B), (Copy_A:-B) ),

        \+(B),

      xgen_trees(not__(B), not__(Gen_B) , Proof, Gen_proof,GenIds),
      gen_conjoined_set(not__(A), Gen_A,  Rest, RestProofs,
                    RestGenProofs, RestGenIds) .

/* Body of unfolded clause is itself unshielded

xgen_trees(not__(A), not__(Gen_A) ,
      [Neg_Head_Id, [not__(A),[Neg_Bod_Id, B]]],
      [Neg_Head_Id,  [not__(Gen_A), [Neg_Bod_Id,Gen_B ]]],
        [Neg_Head_Id,  [ [Neg_Bod_Id ]]]) :-

         \+(A = not__(_)),

        clause_impress(Gen_A, Gen_B, Head_Id, unshielded),

      Neg_Head_Id is -Head_Id,
         copy( (Gen_A:-Gen_B), (A:-B) ),
      \+(B),
           clause_impress( Gen_B, _, Bod_Id, shielded),
          tell(user),
```

```
            nl, write('unfolding - shielded body, case 4a'), nl,
            told,
          Neg_Bod_Id is -Bod_Id.
*/

/* check for predicates which are never true - undefined */

xgen_trees(not__(Head) ,not__(Gen_head),[Neg_Clause_ID_no,
              not__(Head)],[Neg_Clause_ID_no, not__(Gen_head)],
                          [Neg_Clause_ID_no]):-
        \+(Head),
        clause_impress(Gen_head, fail, Clause_ID_no,_),      tell(user),
        nl, write('negative fact, undefined predicate - Case 4a'), nl,
        told,
        Neg_Clause_ID_no is -Clause_ID_no,
        !.

xgen_trees(not__(A), not__(Gen_A) ,  Proof,  Gen_proof, GenIds) :-
        (A = not__(B)), (Gen_A =  not__(Gen_B)),      tell(user),
        nl, write('Case 4a, not not A = A '), nl,
        told,
        xgen_trees(B, Gen_B ,  Proof,  Gen_proof, GenIds).


/*case 4a : A shielded - no unfolding */
/* The parts of the proof concerning either built-in or shielded
predicates are not included in the proof tree. */

xgen_trees(not__(A), not__(Gen_A), [Neg_Clause_Id, not__(A)],
[Neg_Clause_Id, not__(Gen_A)], [Neg_Clause_Id]) :-
          \+(A = not__(_)),
                \+(A),
           clause_impress(Gen_A, _,  Clause_Id, shielded),
         tell(user),
       nl, write('case 4a, A shielded -- no unfolding '), nl,
        told,
        Neg_Clause_Id is -Clause_Id.


xgen_trees(not__(A), not__(Gen_A), [0, not__(A)], [0, not__(Gen_A)] , [0]) :-
        built_in(A),
        !,      tell(user),
            nl, write('Case 4a, negative built in'), nl,
        told,
        \+(A).

/*********

Case 5 Expr is a negated conjunction: this is expanded  into a
disjunction

*********/

xgen_trees(not__((A , _)), not__((Gen_A ,_) ), Proof, Gen_proof,GenIds) :-
        not__(A),
        tell(user),
        nl, write('Case 5, expanding negated conjunction'), nl,
```

```
        nl, write('first arm succeeds'), nl,
        told,
        xgen_trees(not__(A), not__(Gen_A), Proof, Gen_proof, GenIds),
        !.


xgen_trees(not__((A , B)), not__((Gen_A , Gen_B)), Proof, Gen_proof, GenIds) :-
        A, not__(B),      tell(user),
        nl, write('Case 5, expanding negated conjunction'), nl,
        nl, write('first arm fails, second arm succeeds'), nl,
        told,
        told,
              xgen_trees( A, Gen_A , A_Proof, Gen_A_proof, Gen_A_Ids),
        xgen_trees(not__(B), not__(Gen_B), Neg_B_Proof,
                   Neg_B_Gen_proof,Gen_B_Ids),
 /* proof of A is appended to that of not__(B) */
        append(A_Proof, Neg_B_Proof, Proof),
        append(Gen_A_proof, Neg_B_Gen_proof, Gen_proof),
              append(Gen_A_Ids, Gen_B_Ids, GenIds),
        !.

/*******
 Case 6 expands a negated disjunction into a conjunction

*******/

xgen_trees(not__((A ; B)), not__((Gen_A ; Gen_B)), Proof, Gen_proof,GenIds) :-
               not__(A),
               not__(B),      tell(user),
              nl, write('Case 6, expanding disjunction'), nl,
                told,
               xgen_trees(not__(A), not__(Gen_A), A_Proof,
                          Gen_A_Proof,Gen_A_Ids ),
               xgen_trees(not__(B), not__(Gen_B), B_Proof,
                          Gen_B_Proof,   Gen_B_Ids ),
               append(A_Proof, B_Proof, Proof),
               append(Gen_A_Proof, Gen_B_Proof, Gen_proof),
                     append(Gen_A_Ids, Gen_B_Ids, GenIds),
               !.

/* ground-ness check */

/* ground_not__ goals are checked - that they are ground. Otherwise
this type of negation is treated as with not__ Goals
*/
xgen_trees(ground_not__(Head), ground_not__( Gen_head),
              Proof, GenProof, GenIds) :-
       ( ground(Head);
          (tell(user),nl,
           write('calling not/1 with nonground argument:'), nl,
           write(Head), nl, told)),

         xgen_trees(not__(Head), not__( Gen_head),Proof, GenProof, GenIds),
        !.

/* predicate which generates generalised goal */
```

```
copy(Old,New):- assert('$marker'(Old)),
                retract('$marker'(New)).

/****************************************************************/


/* rename(:X, :Y).

See pages 167-174 Sterling and Shapiro .
(NB Sicstus library(terms) might also be useful.)

The assumption is that X  is more general than Y. The code renames
variables from Y to  match with X.
*/
rename(X,Y) :- var(X), var(Y),  X=Y.
rename(X,Y) :- nonvar(X),nonvar(Y),constant(X),constant(Y), X=Y.
rename(X,Y) :- nonvar(Y), var(X).
rename(X,Y) :- nonvar(X),nonvar(Y),compound(X),compound(Y),

               term_rename(X, Y).

   term_rename(X, Y) :- functor(X,F,N), functor(Y,F,N), rename_args(N,X,Y).
   rename_args(N,X,Y) :- N > 0, rename_arg(N, X, Y), N1 is N-1,
       rename_args(N1,X,Y).

   rename_args(0,X,Y).

   rename_arg(N, X, Y ):-
      arg(N, X, ArgX), arg(N, Y, ArgY), rename(ArgX,ArgY).



 constant(X) :- atom(X);number(X).

/****************************************************************/

flatten_list([],[]).

flatten_list([H|T],[H|Flattened_tail]):-
        \+ list(H),
        flatten_list(T,Flattened_tail).

flatten_list([H|T],F):-
        flatten_list(H,Flattened_head),
        flatten_list(T,Flattened_tail),
        append(Flattened_head,Flattened_tail,F).

list([H|T]).

/*
 flatten_list_of_atomics(List,Flattened_list) accepts a list of
 ATOMIC elements, List, and flattens it to form a list,
 Flattened_list.

 Test Data

  flatten_list_of_atomics([[2]],X).
```

```
  flatten_list_of_atomics([[a]],X).
  flatten_list_of_atomics([[[[a]]]],X).
  flatten_list_of_atomics([a,[b,[c],d],e,[f]],X).
  flatten_list_of_atomics([a,[a]],X).

  NB flatten_list_of_atomics([[a(b)]],X). fails

 C.B. 16/10/96

*/

flatten_list_of_atomics([],[]).

flatten_list_of_atomics([H|T],[H|Flattened_tail]):-
        atomic(H),
        flatten_list_of_atomics(T,Flattened_tail).

flatten_list_of_atomics([H|T],F):-
        flatten_list_of_atomics(H,Flattened_head),
        flatten_list_of_atomics(T,Flattened_tail),
        append(Flattened_head,Flattened_tail,F).

/*******************************************/


built_in(call(_)).

built_in(!).
built_in(number(_)).
built_in(_=_).
built_in(_<_).
built_in(_>_).
built_in(_ is _).
built_in(_=.._).
built_in(write(_)).
built_in(nl).
built_in(_ == _).
built_in(_ =< _).
built_in(_ >= _).
%%built_in(not__(_)).
built_in(\+(_)).
%%built_in(ground_not__(_)).

/*****************************************************************/

append([],L,L).
append([H|T],L,[H|Z]) :- append(T,L,Z).

/*****************************************************************/

/* form_list_of_rule_IDs(L, List_of_rule_IDs) accepts a list, L, of
lists, where each sublist represents a rule. The sublists for rules
take the form

        [Rule_ID, Head_of_rule, Subgoal_1, Subgoal_2.... ]

The procedure outputs a list of the IDs for the rules.
```

```
Tests...
form_list_of_rule_IDs([[8,c,c1],[10,d,d1]], [8, 10]).


 */
form_list_of_rule_IDs([],[]).

form_list_of_rule_IDs([ [H|_] | T2], [H|T2_IDs]):-
        form_list_of_rule_IDs(T2, T2_IDs).

/*****************************************************************/

/* rule_counter([1,1,1,1,2,3,1,1,1, 2],
          [potential(1,7),
           potential(2,2),
           potential(3,1)
          ]).
*/

rule_counter([],[]).

rule_counter([H|T],[potential(H,N)|Rest]):-

        del(H,T,L),
        count(H,T,NT),
        N is NT + 1,
        rule_counter(L,Rest).

/*****************************************************************/


                    /* utilities */

list([_|_]).

/*****************************************************************/

append([],L,L).
append([H|T],L,[H|Z]) :- append(T,L,Z).

/*****************************************************************/
/* This version of remove_dups retains original order of list but is
inefficient */

/*test
remove_dups([a,a,a,a,b,c,a,a,a, b], X).

X = [a,b,c] ?

remove_dups([1,1,1,1,2,3,1,1,1, 2], X).

X = [1,2,3] ?

*/

remove_dups([X|Xs], [X|Ys]) :-  del(X, Xs, Xs1),
```

```
        remove_dups( Xs1,  Ys).

remove_dups([], []).

/*****************************************************************/

/* del(X, Y, Z) deletes all occurences of X from Y yielding Z */
del( X, [X|Xs], Ys) :- del( X, Xs, Ys).

del( Z, [X|Xs], [X|Ys]) :- \+(X  = Z),

                        del( Z, Xs, Ys).

del( _, [], []).

/*****************************************************************/

/* count(Item, List, Count) counts the number of occurrences of Item
in the List.

count(1, [1,1,1,1,2,3,1,1,1, 2], 7).
count(2, [1,1,1,1,2,3,1,1,1, 2], 2).

 */

count(_,[],0).

count(H,[H|T],N):-
      count(H,T,NT),
      N is NT + 1.

count(X,[H|T],N):-
      \+(X=H),
      count(X,T,N).

%%
%%
%% partition(Divider, List1, List2, List3)
%%
%%

partition(Divider, [H | T], [H | S1], S2):-

   H =< Divider,

   partition(Divider, T, S1, S2).

partition(Divider, [H | T], S1, [H | S2]):-

   H > Divider,

   partition(Divider, T, S1, S2).

partition(_, [], [], []).

%%
%%
```

```
%% qsort(List1, List2)
%%
%%

qsort( [], [] ).

qsort( [H | T], Slist ):-

    partition( H, T, L1, L2 ),

    qsort( L1, Slist1 ),

    qsort( L2, Slist2 ),

    append( Slist1, [H | Slist2], Slist ).
```

F Code for Computing S

```
/*

This code has been designed with the intension that it should be able
to work with Prolog containing:- (CB 31/1/97)

built-in predicates
shielding
conjuctions
disjunctions (procedures and ;)


Algorithm
———

1 For each instance
   form list of IDs for unshielded rules that succeed
   remove duplicates from this list
2 Append together all the uniqued ID-lists for the instances and sort
the new list into ascending order.
3 Remove duplicates from the sorted list, annotating it with counts.

NB Duplicate rules are identified using the ID.

NB clause_impress/4 facts are assumed to exist prior to execution of
the code in this file.

*/


/*****************************************************************/

/*
          list_S_rules(List_of_instances, List_of_S_rules)

Currently, this is the top level procedure.

test with

list_S_rules([kill(john,john), patricide(john,father(john))],
Answer).

*/

list_S_rules(List_of_instances, List_of_S_rules):-

       get_S_rules(List_of_instances,L),
       qsort(L,Sorted_L),
       rule_counter(Sorted_L, List_of_S_rules).

/*****************************************************************/

/* test with
get_S_rules([kill(john,john), patricide(john,father(john))], List_of_rules).
*/
```

```
get_S_rules([],[]).
get_S_rules([H|T], Rules):-
        get_IDs_of_S_rules(H, Hrules),
        get_S_rules(T, Trules),
        append(Hrules, Trules, Rules).

/*****************************************************************/

/* test with
get_IDs_of_S_rules(kill(john,john), List_of_rules).
*/

get_IDs_of_S_rules(Instance, List2):-
        assertz(list_of_IDs([])),
        assert_IDs_S(Instance),
        list_of_IDs(List),
        remove_dups(List, List2),
        abolish(list_of_IDs).

/* The list of IDs is asserted to Prolog's memory, as opposed to being
passed back by the lower level procedures, because one of these
procedures, namely assert_IDs_of_S_rules, will (CORRECTLY)
fail whenever it is passed a Goal which would fail if it were a Prolog
query. */

/*****************************************************************/

assert_IDs_S(Instance):- assert_IDs_of_S_rules(Instance).

assert_IDs_S(_). % ensures that this procedure never fails.

/* The purpose of this procedure is to prevent the rule
get_IDs_of_S_rules failing. The procedure
assert_IDs_of_S_rules will (CORRECTLY) fail whenever it is
passed a Goal which would fail if it were a Prolog query. */

/*****************************************************************/

assert_IDs_of_S_rules( (Goal ; _) ) :-
        assert_IDs_of_S_rules(Goal).

assert_IDs_of_S_rules( (_ ; Goal) ) :-
        !,
        assert_IDs_of_S_rules(Goal).

/* The first two clauses concern disjunctions by ; */


assert_IDs_of_S_rules( (Goal1, Goal2) ) :-
        !,
        assert_IDs_of_S_rules(Goal1),
        assert_IDs_of_S_rules(Goal2).

/* The clause above concerns conjunctions. */
```

```
assert_IDs_of_S_rules(true) :- !.

assert_IDs_of_S_rules(Undefined_predicate) :-
        clause_impress(Undefined_predicate, fail, _, undefined),
        !,
        fail.

assert_IDs_of_S_rules(Goal) :-
        built_in(Goal),
        Goal,
        !,                  % NB Built-ins are never redone.
        add_ID_to_list(0).  % Built-ins are given the ID 0.

assert_IDs_of_S_rules(Goal) :-
        built_in(Goal),
        !,
        fail.

assert_IDs_of_S_rules(Goal) :-
        \+ built_in(Goal),
        clause_impress(Goal, Body, Clause_ID, Shield_status),
        process_wrt_shield_status_S(Shield_status, Body, Clause_ID),
        !.

/****************************************************************/

process_wrt_shield_status_S(_, true, _).

process_wrt_shield_status_S(shielded, Body, _):- Body.

/* Subgoals arrising from shielded predicates are not traced. */

process_wrt_shield_status_S(unshielded, Body, Clause_ID):-
        assert_IDs_of_S_rules(Body),
        add_ID_to_list(Clause_ID).

/****************************************************************/

add_ID_to_list(ID):-
        retract(list_of_IDs(ID_list)),
        append([ID], ID_list, Updated_ID_list),
        assertz(list_of_IDs(Updated_ID_list)).

/****************************************************************/

built_in(call(_)).

built_in(!).
built_in(number(_)).
built_in(_=_).
built_in(_<_).
built_in(_>_).
built_in(_ is _).
built_in(_=.._).
built_in(write(_)).
built_in(nl).
built_in(_ == _).
```

```
built_in(_ =< _).
built_in(_ >= _).
built_in(not__(_)).
built_in(\+(_)).
built_in(ground_not__(_)).

/******************************************************************/


/* rule_counter([1,1,1,1,2,3,1,1,1, 2],
          [potential(1,7),
           potential(2,2),
           potential(3,1)
          ]).
*/

rule_counter([],[]).

rule_counter([H|T],[potential(H,N)|Rest]):-

        del(H,T,L),
        count(H,T,NT),
        N is NT + 1,
        rule_counter(L,Rest).

/******************************************************************/


                    /* utilities */

/******************************************************************/

append([],L,L).
append([H|T],L,[H|Z]) :- append(T,L,Z).

/******************************************************************/
/* This version of remove_dups retains original order of list but is
inefficient */

/*test
remove_dups([a,a,a,a,b,c,a,a,a, b], X).

X = [a,b,c] ?

remove_dups([1,1,1,1,2,3,1,1,1, 2], X).

X = [1,2,3] ?

*/

remove_dups([X|Xs], [X|Ys]) :-  del(X, Xs, Xs1),
          remove_dups( Xs1,  Ys).

remove_dups([], []).

/******************************************************************/
```

```
/* del(X, Y, Z) deletes all occurences of X from Y yielding Z */
del( X, [X|Xs], Ys) :- del( X, Xs, Ys).

del( Z, [X|Xs], [X|Ys]) :- \+(X  = Z),

                           del( Z, Xs, Ys).

del( _, [], []).

/*****************************************************************/

/* count(Item, List, Count) counts the number of occurrences of Item
in the List.

count(1, [1,1,1,1,2,3,1,1,1, 2], 7).
count(2, [1,1,1,1,2,3,1,1,1, 2], 2).

 */

count(_,[],0).

count(H,[H|T],N):-
        count(H,T,NT),
        N is NT + 1.

count(X,[H|T],N):-
        \+(X=H),
        count(X,T,N).

%%
%%
%% partition(Divider, List1, List2, List3)
%%
%%

partition(Divider, [H | T], [H | S1], S2):-
    H =< Divider,
    partition(Divider, T, S1, S2).

partition(Divider, [H | T], S1, [H | S2]):-
    H > Divider,
    partition(Divider, T, S1, S2).

partition(_, [], [], []).

%%
%%
%% qsort(List1, List2)
%%
%%

qsort( [], [] ).

qsort( [H | T], Slist ):-
    partition( H, T, L1, L2 ),
    qsort( L1, Slist1 ),
    qsort( L2, Slist2 ),
```

```
append( Slist1, [H | Slist2], Slist ).
```