Transforming general program proofs: a meta interpreter which expands negative literals

West, MM, Bryant, CH and McCluskey, TL

Title	Transforming general program proofs: a meta interpreter which expands negative literals
Authors	West, MM, Bryant, CH and McCluskey, TL
Publication title	
Publisher	
Type	Conference or Workshop Item
USIR URL	This version is available at: http://usir.salford.ac.uk/id/eprint/1768/
Published Date	1997

USIR is a digital collection of the research output of the University of Salford. Where copyright permits, full text material held in the repository is made freely available online and can be read, downloaded and copied for non-commercial private study or research purposes. Please check the manuscript for any further copyright restrictions.

For more information, including our policy and submission procedure, please contact the Repository Team at: library-research@salford.ac.uk.

Transforming General Program Proofs: A Meta Interpreter which Expands Negative Literals

M.M. West, C.H.Bryant, T.L. McCluskey

School of Computing and Mathematics,
The University of Huddersfield,
HD1 3DH,
UK.

Fax: +44-1484-421106
Email: impress@hud.ac.uk
URL: http://www.hud.ac.uk/schools/comp+maths/home.html

The IMPRESS project is supported by an EPSRC grant, number GR/K73152.

Abstract

This paper provides a method for generating a proof tree from an instance and a general logic program, viz. one which includes negative literals. The method differs from previous work in the field in that negative literals are first unfolded and then transformed using De Morgan's laws, so that the tree explicitly includes *negative* rules.

The method is applied to a real-world example, a large executable specification providing rules for separation for two aircraft. Given an instance of a pair of aircraft whose flight paths potentially violate seperation rules, the tree contains both positive and negative rules which contribute to the proof.

1 Introduction

Logic programming languages such as Prolog include features for implementing meta-programs. Consequently they may be used to implement meta-interpreters which generate proof trees, that is hierarchic structures which represent proofs of successful queries.

Proof trees are important to logic program debugging, logic program analysis and explanation-based generalisation (EBG) [1], a technique of machine learning which requires that a generalised proof tree be generated.

A simple approach to debugging a logic program is to execute a test-set of queries, where the desired outcome of each query is known prior to its execution. 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. 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. Furthermore parts of a logic program may be unreachable, that is they will never be executed whatever the query. More sophisticated strategies which use proof trees are needed to find such errors.

The 'text-book' meta-interpreters for generating proof trees [2, 3] are restricted to definite programs; they cannot cope with general programs, that is programs which include negative literals. This restriction

severely limits their application because many logic programs contain negation, including the real-world example described later in this paper. As far as these authors are aware (see Section 6), this is the first paper to describe a meta-interpreter for generating proof trees which explicitly represent *negative* rules.

The remainder of the paper is structured as follows. Section 2 lists some definitions and denotation. Section 3 explains how the meta-interpreter unfolds and then transforms negative literals. Section 4 describes how these ideas can be implemented in a logic programming language. Section 5 illustrates the advantage of explicitly representing *negative* rules for a 'real-world' application. Relevant previous work is reviewed in Section 6 which is followed by the Conclusion.

2 Preliminaries

We assume the standard logic program terminology, where a *general* clause 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 $\forall(\mathcal{F})$ denotes the universal closure of $\mathcal{F}$, where all its free variables are universally quantified. In a similar manner, $\exists(\mathcal{F})$ denotes the existential closure of $\mathcal{F}$.

2.1 Negation as Failure and Completion

SLD-resolution allows the derivation of positive consequences (namely, conjunctions of atoms) [4] 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 [5, 6].) In order to justify the use of negation as failure rule, Clarke [7] introduced the idea of the completion of a general program and this is outlined as follows.

The *completed* definition of predicate p requires a new predicate '=' whose intended interpretation is identity. Suppose predicate p ($\in$ *Prog*) 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 series of disjoined predicates of the form:

$$\forall\, \mathbf{x}(p(\mathbf{x}) \longleftrightarrow A_1 \vee \ldots A_m),$$

where each A_i has the general form $\exists\, \mathbf{y_i}(\mathbf{x} = \mathbf{t_i}) \;\wedge\; W_i$, 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'.) This might occur in a program automatically generated from a requirements specification.

For SLDNF resolution, positive literals are 'deleted' via resolution. The proposed solution [4] 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 linked to the main tree via a function $subs(n)$.

Safe negation: sufficient rules for safe negation are *either* that the negative goal must be ground when called *or* that negated goals are in the form $\neg\, \exists\, p(\mathbf{x}, \mathbf{y})$, where non-ground variables $\mathbf{x}$ are bound by the existential quantifier. For a full discussion of this topic see [4, 5].

3 Proof Tree Generation

3.1 Clause Shielding

A proof tree need not include *all* the clauses involved in a proof of an instance, for some clauses can be 'shielded' (this is related to the choice of 'operational' predicates in the EBG literature.) In the work described here, shielded clauses are

1. 'definitional' predicates, whose proof is not required. (We assume a hierarchy where shielded clauses have only shielded clauses in their bodies.) For a given predicate, rules associated with it are either *all* shielded or *all* unshielded. In the case study presented in Section 5.1 these are derived from auxiliary or domain axioms.

2. predicates 'built-in' by the Prolog system, such as 'is', '<', etc.

3.2 Tree Generation with Negative Literals Shielded

First we consider the case where *negation* is shielded, in addition to 1 and 2 above. The definition presented here is based on 'traditional' EBG tree generation described in [8]. 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 [9], 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 3.2.1 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: clause $p(\mathbf{t})$ can be shielded or unshielded. If $p(\mathbf{t})$ is shielded, then it is eliminated via resolution using other shielded clauses and *gen_tree* calls itself recursively with goal argument $(\mathcal{L}, \mathcal{R})$ and node m. If clause $p(\mathbf{t})$ 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 *gen_tree*(G, n) is defined as:

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

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.

3.2.1 Example – Tree1

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.

```
\* clauses numbered from 101 to 104 *\
r(A) :- t(A), not__(p(A,Y)).

p(A, 1) :- m(A, X), Y is 12*X, Y < 20000 .
```

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

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

t(a). t(b).
```

Two kinds of tree are generated, one of which represents a proof of the given instance, the other representing a generalisation of it. 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(X), P, GenP).

P = [101,[r(b),[0,not__(p(b,_A))]]],
GenP = [101,[r(X),[0,not__(p(X,_B))]]] ?
```

3.3 Tree Generation Including Expanded Negative literals

In this section we consider the case where negation is not shielded. The philosophy of this method is that the Negation as Failure rule and subsequent necessity of subsidiary failed trees is 'pushed down' to the 'shielded' rules. A tree is generated which explicitly identifies failed rules involved in the proof of the instance. For each branch, the tree generation continues until *either* the proof associated with the branch is completed, *or* when the clauses concerned are 'shielded'.

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 example code presented above, a call to the tree generator will provide the following response:

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

P = [101,[r(b),[-102,[not__(p(b,1)),[0,not__(36000<24000)]],
                -103,[not__(p(b,2)),[0,not__(36000<36000)]],
                -104,[not__(p(b,3)),[0,not__(36000<20000)]]]]],

GenP = [101,[r(X),[-102,[not__(p(X,1)),[0,not__(_A<24000)]],
                   -103,[not__(p(X,2)),[0,not__(_B<36000)]],
                   -104,[not__(p(X,3)),[0,not__(_C<20000)]]]]] ?
```

As can be seen each of the negated clauses is expanded out and is represented in the tree. *Negated* clauses are provided with a negated identity number. Since there are three clauses with predicate head p, all contribute to the proof.

The tree is constructed as follows. The difference between this and the previous tree is that goals can take the form $(\mathcal{L}, \neg\,(\mathcal{E}), \mathcal{R})$, as well as $(\mathcal{L}, p(\mathbf{t}), \mathcal{R})$, where $\mathcal{E}$ is a conjunction of literals.

We first suppose a goal $(\mathcal{L}, \neg\, q(\mathbf{t}), \mathcal{R})$. Thus suppose node m is a child of n on a successful branch derived with clause $\neg\, q(\mathbf{t})$, (viz. $q(\mathbf{t})$ has failed), where the completed definition of $q(\mathbf{x})$ is written in the form

$$\begin{aligned} q(\mathbf{x}) &\longleftrightarrow \exists\,\mathbf{y_1}(\mathbf{x} = \mathbf{t_1})q(\mathbf{t_1}) \vee \ldots \vee \exists\,\mathbf{y_k}(\mathbf{x} = \mathbf{t_k})q(\mathbf{t_k}) \\ &\longleftrightarrow \textstyle\bigvee_i \exists\,\mathbf{y_i}(\mathbf{x} = \mathbf{t_i}) \;\wedge\; W_i \end{aligned}$$

(The set of $q(\mathbf{t_i})$ correspond to different clause heads matching $q(\mathbf{x})$.) We are assuming that each of the clause heads is from an unshielded clause. We have

$$\begin{aligned}
&\neg\,(\exists\,\mathbf{y_1}(\mathbf{x}=\mathbf{t_1})q(\mathbf{t_1}) \vee \ldots \vee \exists\,\mathbf{y_k}(\mathbf{x}=\mathbf{t_k})q(\mathbf{t_k}))\\
&\quad\longleftrightarrow \neg\,\exists\,\mathbf{y_1}(\mathbf{x}=\mathbf{t_1})(q(\mathbf{t_1})) \wedge \ldots \wedge \neg\,\exists(\mathbf{x}=\mathbf{t_k})\mathbf{y_k}(q(\mathbf{t_k}))\\
&\quad\longleftrightarrow \textstyle\bigwedge_i \neg\,\exists\,\mathbf{y_i}(\mathbf{x}=\mathbf{t_i}) \ \wedge\ W_i.
\end{aligned}$$

The above unfolding process is the first step in the expansion of the negative tree. The process stops only when a component clause is *shielded*. For goals of the form $(\mathcal{L}, \neg\ q(\mathbf{t}), \mathcal{R})$ the negated literal is replaced by the equivalent $\bigwedge_i \neg\,(\exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_k})\mathbf{y_i}(q(\mathbf{t_i})))$.

The introduction of the $\exists$ is because the 'not' is a test and the instantiation of variables does not affect $\mathcal{L}, \mathcal{R}$. 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 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 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 $\bigwedge_j L_j$. Then

$$\begin{aligned}
&\neg\,(\exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_i}) \ \wedge\ \textstyle\bigwedge_j L_j) \longleftrightarrow \neg\ \exists\,\mathbf{y_i} \textstyle\bigwedge_j (\mathbf{t}=\mathbf{t_i})L_j\\
&\quad\longleftarrow \textstyle\bigvee_j (\neg\ \exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_i})L_j).
\end{aligned}$$

(Note that the converse is not necessarily true.)

It is only necessary for *one* of the L_j to fail for W_i to fail, and there may be more than one sub-tree associated with the failure of each W_i. We thus consider one of the sub-trees and we suppose this to be the one associated with L_j; we assume that L_j 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 $\mathcal{L}, (\neg\ \exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_i}) \ \wedge\ L_j), \mathcal{R}$.

An exception is the case where one or more of the disjoined literals is itself negative. Thus suppose that L_j is of the form $\neg\ M$, then the goal becomes $\mathcal{L}, M, \mathcal{R}$.

Hence the tree *gen_tree*(G, n) is defined as:

$$\begin{aligned}
\textit{gen_tree}(G,n) &= \neg\ q(t), \textit{gen_tree}((\mathcal{L},\mathcal{R}),m) : p(t)\ \text{is}\ \neg\ q(t)\ \text{and}\ q(t)\ \text{is shielded}; && (4)\\
&= \bigwedge_i \textit{gen_tree}(\mathcal{L},(\neg\ \exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_i}) \ \wedge\ W_i),\mathcal{R}),m) : p(t)\ \text{is}\ \neg\ q(t); && (5)\\
&= \textit{gen_tree}((\mathcal{L},(\neg\ \exists\,\mathbf{y_i}(\mathbf{t}=\mathbf{t_i}) \ \wedge\ L_j),\mathcal{R}),m) : p(t)\ \text{is}\ \neg \bigwedge_i L_i\ \text{and}\\
&\qquad\qquad L_j\ \text{is a positive literal which fails} && (6)\\
&= \textit{gen_tree}(\mathcal{L},M,\mathcal{R}) : p(t)\ \text{is}\ \neg\ (\neg\ M) && (7)
\end{aligned}$$

4 Implementation

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 rules associated with a given predicate. (This is true of the case study in Section 5.1.) The tree output consists of the identity number of each rule, together with the rules themselves. If a rule 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 3.3.

In *gen_tree* definition (6) above we need to obtain proofs of expressions such as $\neg \bigwedge_i E_i$. Expanding to obtain the disjunction $\bigvee_i(\neg E_i)$, we may find that (since Prolog uses a left-to-right computation rule) a given component is insufficiently instantiated. We thus recursively replace :

$$\neg (E_i) \vee \neg (E_{i+1}) \longleftrightarrow \neg (E_i) \vee ((E_i) \wedge \neg (E_{i+1})).$$

5 Application to a Large Case Study

5.1 CPS

The case study is derived from part of the ongoing work of the IMPRESS project[1] [10]. The aim of the project is the improvement of an existing formal requirements specification using methods from machine learning such as explanation-based generalisation and theory revision. The existing specification is a 'conflict prediction specification' (CPS) for the control of aircraft flying in the eastern half of the North Atlantic. The requirements, written in Many Sorted First Order Logic, consists of a theory of over 1,000 axioms, held in a tools environment supporting validation. (The development and validation of the existing CPS is described in [11].)

Two of the tools components were a parser for identifying syntactic errors, and a prototyping tool for generating an executable form of the CPS in Prolog. Batches of expert-derived test cases were used to compare expected and actual results. Tests take 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 English. 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 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 [12], do not accept as input theories containing negation.

The current version of the CPS has been translated to sicstus Prolog, and the translation gives rise to clauses corresponding to main axioms, and those (definitional) corresponding to auxiliary axioms and domain objects. Both main and auxiliary contain *general* clauses, which allow negative literals in their bodies. The translation mechanism deals with negation in two ways. Expressions which result in clauses of the form $\neg \exists z(q(z))$, are translated to 'is not provable', viz. \+ in sicstus Prolog. For all other forms of negation, the goal must be ground, and this is checked. The executable form of the CPS is complex, containing 50 unshielded rules, 250 auxiliary and domain object rules, and over 1000 facts concerned with aircraft, airfields and flight plans.

5.2 Application to CPS

The target concept is of a pair of aircraft whose flight plans are 'in conflict'. An instance of a pair of flight plans is provided, together with aircraft identifiers, aircraft types etc. The flight plans involve flight paths (sequences of flight segments) with latitudinal and longitudinal co-ordinates and flight levels. Given the instance and concept goal, a proof is given of the conflicting flight plans. The following is a tiny fragment of the rule tree, where rules are represented by numbers. A pictorial representation of this fragment is shown in Figure 1.

[1] IMProving the quality of formal REquirements SpecificationS

```
GenProof =  [1,[[100056,100043,2, [[7, [[3, ...,10076]], 8,[[10075]], ..

                        [[-30,[[-31,[[-10084,]],-32,[[-1042,]]]]]],..]] ]] ]
```

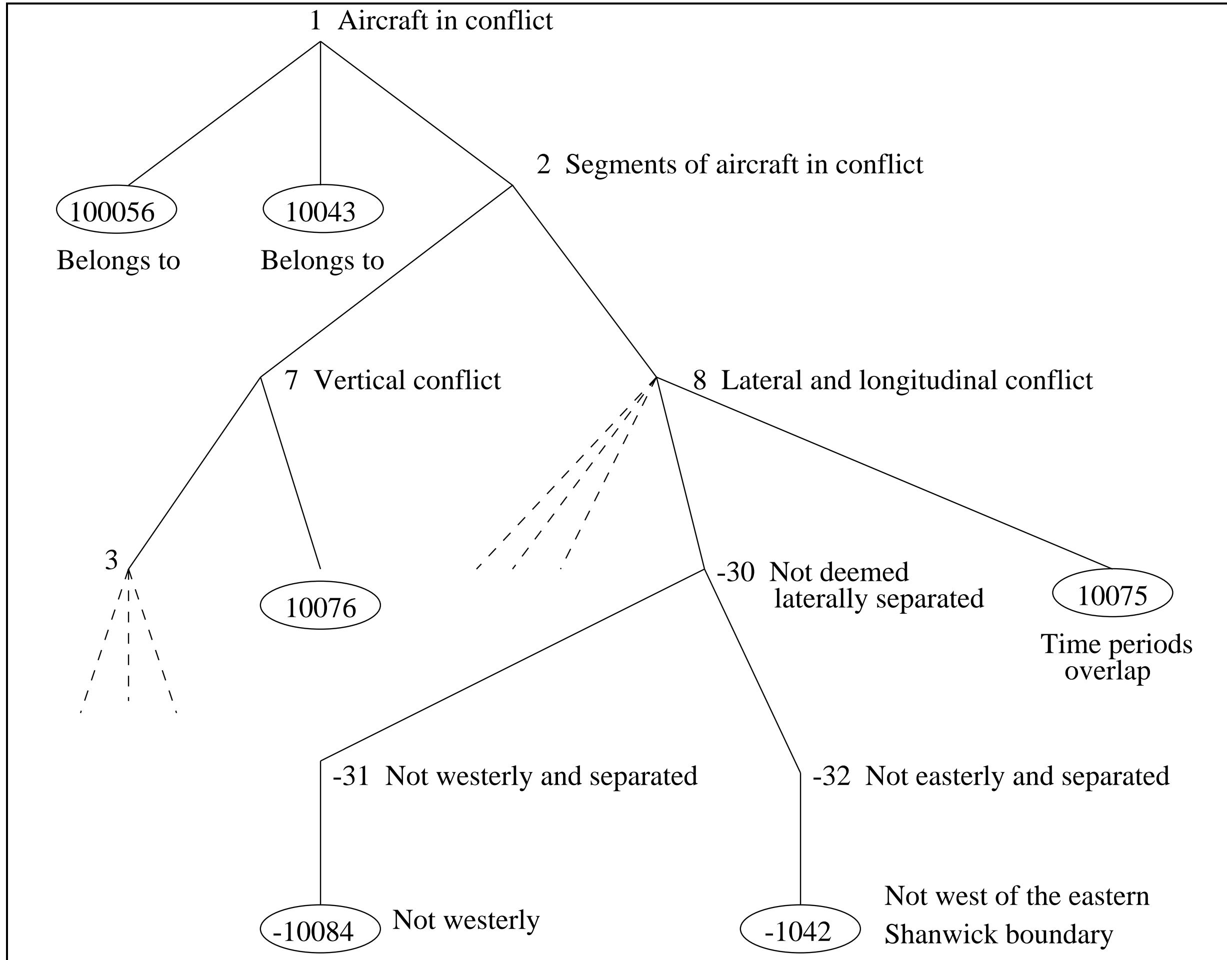

Figure 1: A Proof Tree Fragment.

6 Previous Work

Siqueira and Puget have described a method for generating a failed proof tree ([13]), 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 [14, 9]: EBGF has been used to aid the generation of trees for proofs which use SLDNF-resolution. General clauses can be associated with the tree. ([9] also includes a review of other methods.)

6.1 EBGF - method

The method uses the definition of program completion (Section 2.1) 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.

6.2 EBGF - extension

The method is extended by Schrödel in [14, 9], using traditional EBG described in [8]. 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

$$ebgf(G, n) = p(\mathbf{t}), \bigwedge_i ebgf(((\mathcal{L}, \mathcal{R}), n_i) : p(\mathbf{t}) \text{ is shielded}; \quad (8)$$

$$= \bigvee_i (\mathbf{t} = \mathbf{t_i}) ebgf((\mathcal{L}, B_i, \mathcal{L}), n_i) : p(\mathbf{t}) \text{ is unshielded} \quad (9)$$

The *ebgf* tree is joined to the main tree via the function $subs(n)$ defined in Section 2.1. 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 [5].

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.

7 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 this is the first paper to describe such a meta-interpreter. The explicit representation of negative rules is achieved by first unfolding negative literals and then transforming them using De Morgan's laws.

References

[1] T. Ellman. Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, 21(2):164–219, 1989.

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] T.L. McCluskey, J.M. Porteous, M.M. West, and C.H. Bryant. The validation of formal specifications of requirements. In *Proceedings - Northern Formal Methods Workshop , to appear*, September 1996.

[11] 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.

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

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

[14] 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.