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* clauses.

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 clauses which contribute to the proof. This more accurate proof contribution is used for blame assignment, for help to spot errors in the specification.

1 Introduction

The creation of domain models using many-sorted logic places a heavy burden on the *validation* process, that is in the promotion of the accuracy of the model with respect to the reality being modelled. Transformation (of parts) of such a model to a general clausal form program *Prog* is sometimes possible, for the sake of testing and simulation, to help in the validation process [1]. Indeed, it is our contention that validation tools could be enhanced by forms of theory revision, although this depends to a large extent on the accurate blame assignment to erroneous parts of the specification's executable form [2].

Assuming *Prog* is automatically derived from a property - oriented specification this way, it would not (unless the translation program was very sophisticated) resemble a typical logic program in that it may have little if any high level recursion, and, in particular, it may well have many instances of negation in clause bodies [3].

Automated aids to debugging *Prog* (and hence debugging of the original specification) using a large test set tend to rely on the use of a meta-interpreter. However the 'text-book' meta-interpreters for generating proof trees [4, 5] 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. Current research, as in [6, 7] has not extended meta-interpreters for generating proof trees to include general programs. This paper describes a meta-interpreter for generating proof trees which explicitly represent *negative* clauses. (As far as we are aware this is the first paper describing such an extension – see Section 6.)

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 (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 $\mathcal{F}$.

SLD-resolution allows the derivation of positive consequences (namely, conjunctions of atoms) [8] 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 [9, 10].) In order to justify the use of negation as failure rule, Clark [11] 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:

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

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 [8] 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.

3 Proof Tree Generation

3.1 Predicate Shielding

A proof tree need not include *all* the predicates involved in a proof of an instance, for some predicates can be 'shielded' (this is related to the choice of 'operational' predicates in the EBG literature.) No children are generated from shielded predicates and the shielded predicates are a subset of the leaf nodes of the tree. For each branch, the tree generation continues until *either* the proof associated with the branch is completed, *or* when the predicates concerned are shielded. 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. In the case study presented in Section 5 the shielded predicates 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 [12]. 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 [13], 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: 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 *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.

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 (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).
```

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),[114,t(b), 0,not__(p(b,_A))]]],
GenP = [101,[r(X),[114,t(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 our method is that the Negation as Failure rule and subsequent necessity of subsidiary failed trees is 'pushed down' to the shielded predicates. A tree is generated which explicitly identifies failed clauses 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 predicates 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),[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(X),[114,t(X),
           -102,[not__(p(X,2)),[112,m(X,_B),0,
              _A is 12*_B,0,not__(_A<24000)]],
           -103,[not__(p(X,3)),[112,m(X,_D),0,
              _C is 12*_D,0,not__(_C<36000)]],
           -104,[not__(p(X,1)),[112,m(X,_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 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. Since clauses `m(b,3000)`, `36000 is 12*3000` all succeed and contribute to the proof, they are included.

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 3.3.) In order to capture the contribution of all the literals, we include the subset of literals which share variables and which succeed. From Lemma 1, Appendix A, we can show that:

$$\begin{aligned} &\neg\ (\exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i})\ \wedge\ \bigwedge_j L_{ij}) \\ &\quad \longleftarrow (\forall\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i})(\ \bigwedge_{\alpha \in A} L_{i\alpha}) \wedge \bigvee_{\beta \in B}(\neg\ \exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i}) L_{i\beta})) \vee (\neg\ \exists\, \mathbf{y_i}(\mathbf{t} = \mathbf{t_i})\ \bigwedge_{\alpha \in A} L_{i\alpha}) \qquad (\mathbf{N2}). \end{aligned}$$

(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[1]. 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. We impose a further restriction on $\{L_{i\alpha} : \alpha \in A\}$, that each $L_{i\alpha}$ shares variables with L_{ik}. 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:

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

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

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

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. (This is true of the case study in Section 5.) The tree output consists of the identity number of each clause, together with the clauses themselves. 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 3.3.

In *gen_tree* definition (6) above we need to obtain proofs of expressions such as $\neg \bigwedge_i E_i$. In order to implement **(N2)** using Prolog's left-to-right computation rule we recursively replace :

$$\neg (L_j) \vee \neg (\bigvee_i (E_i)) \longleftrightarrow \neg (L_j) \vee ((L_j) \wedge \neg (\bigvee_i E_i)).$$

An example is `m(A,X)` in Section 3.3.

The translation mechanism described in the next section 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. This implementation of negation is safe, as it satisfies the following rule for safe negation: *either* the negative goal must be ground when called *or* the negated goals are in the form $\neg \exists p(\mathbf{x})$, where non-ground variables of $\mathbf{x}$ are bound by the existential quantifier. For a full discussion of this topic see [8, 9].

5 Application to a Large Case Study

The case study is derived from part of the ongoing work of the IMPRESS project[2] [2]. The aim of the project is the improvement of an existing formal requirements specification using methods from machine learning such as theory revision and explanation-based generalisation. 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 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 [1].)

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 [14], do not accept as input theories containing negation. In this case, negated predicates would be automatically shielded; if an error is hidden in a clause called by negation, then no assignment of blame can be made. Hence information which should contribute to the discovery of blame is not being found.

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 unshielded clauses, 250 auxiliary and domain object clauses, and over 1000 facts concerned with aircraft, airfields and flight plans.

[2] IMProving the quality of formal REquirements SpecificationS

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 full trees are presented in Appendices B and C, where clauses are represented by numbers. Appendix C shows the extended version of the tree, and Figure 1 contains a pictorial representation of a fragment of this tree. The numbers in ellipses are from definitional or built in predicates and are shielded. For example '10076'

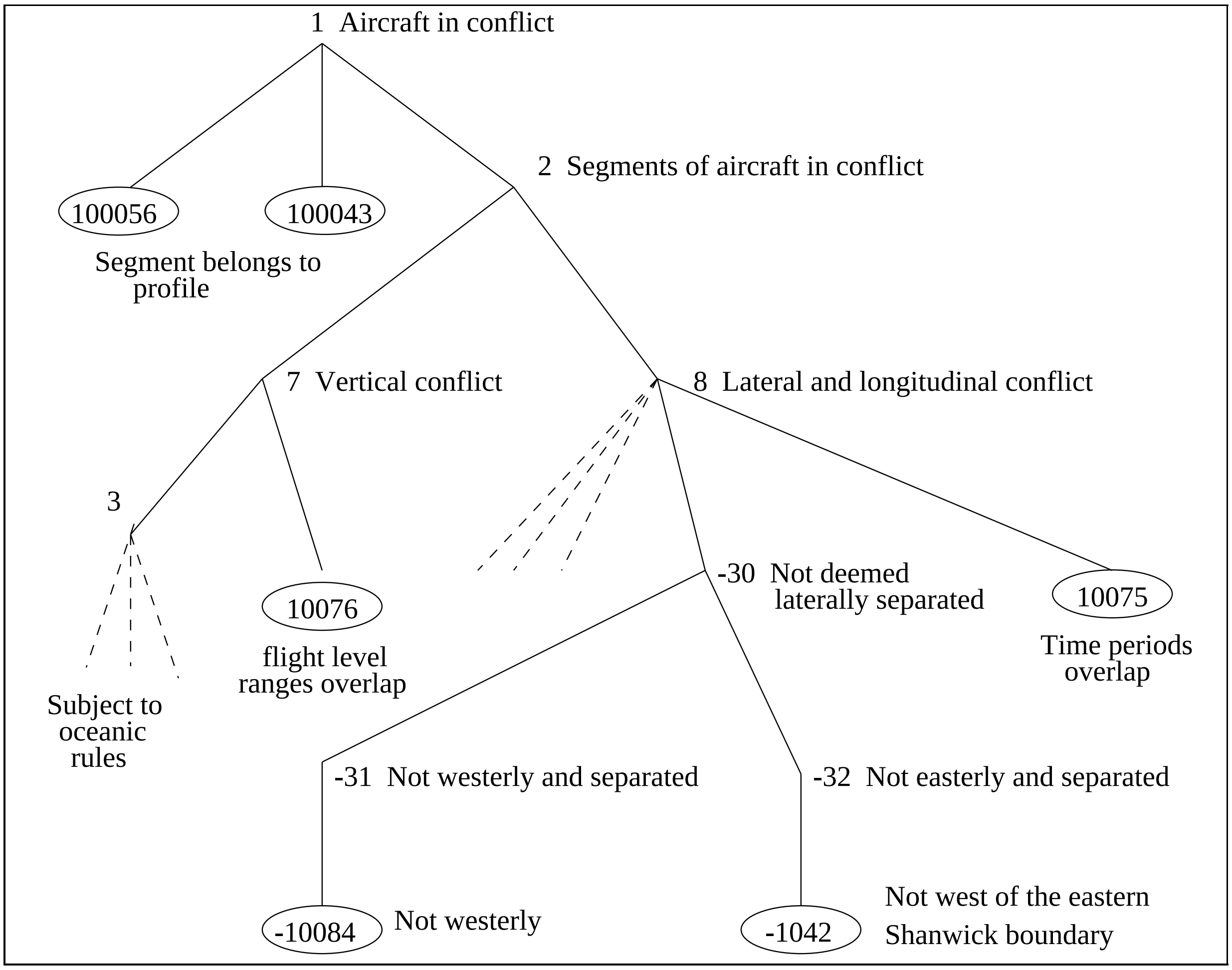

Figure 1: A Proof Tree Fragment.

is the identity of one of the clauses associated with the definition of overlap of the flight level ranges of the two aircraft. '-10084' is the negation of one of the clauses associated with the 'westerly' definition. Tree nodes not in ellipses are associated with unshielded predicates.

6 Previous Work

Siqueira and Puget have described a method for generating a failed proof tree ([15]), 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 [16, 13]: EBGF has been used to aid the generation of trees for proofs which use SLDNF-resolution. General predicates can be associated with the tree. ([13] also includes a review of other methods.)

The method uses the definition of program completion (**N1**, Section 2) 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. Recalling 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.

The method is extended by Schrödel in [16, 13], using traditional EBG described in [12]. 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 a subsidiary function described in Section 2. 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 [9].

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 predicates are redefined and integrated with the successful predicates. Thus negation is 'deferred' to the leaf nodes of the tree. This has the advantage that the failed predicates of interest, viz. the unshielded predicates 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

Current meta-interpreters for generating proof trees have previously been 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 clauses is achieved by first unfolding negative literals and then transforming them using De Morgan's laws.

8 Acknowledgement

The authors would like to thank the referees for a number of helpful comments which have improved this paper.

References

[1] 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.

[2] T.L. McCluskey, J.M. Porteous, M.M. West, and C.H. Bryant. The validation of formal specifications of requirements. In *Proceedings of the BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, September 1996. Electronic Workshops in Computing Series, Springer.

[3] S. Wrobel. First order theory revision. In L De Raedt, editor, *Advances in Inductive Logic Programming, ILP '95* , pages 14–33. IOS Press, 1996.

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

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

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

A Lemma 1: Expansion of Negated term

Lemma 1: $\neg\,(\exists\mathbf{y}\ \bigwedge_{j=1}^{n} E_j) \longleftarrow (\forall\,\mathbf{y}(\ \bigwedge_{\alpha\in A} E_\alpha) \wedge \bigvee_{\beta\in B} \neg\ \exists\mathbf{y} E_\beta) \vee (\neg\ \exists\mathbf{y}\ \bigwedge_{\alpha\in A} E_\alpha)$
where A and B arbitrarily partition $\{1,\ldots,n\}$.
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\text{]}\\
&\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\text{]}\\
&\longleftarrow \forall\,\mathbf{y}(\ \textstyle\bigwedge_{\alpha\in A} E_\alpha \wedge \neg\ \bigwedge_{\beta\in B} E_\beta) \vee \forall\,\mathbf{y}(\neg\ \ \bigwedge_{\alpha\in A} E_\alpha)\\
& && \text{[Distribution of } \forall \text{ over } \vee \text{ -- implied by]}\\
&\longleftarrow \forall\,\mathbf{y}(\ \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge (\forall\,\mathbf{y}\neg\ \bigwedge_{\beta\in B} E_\beta) \vee (\neg\ \exists\mathbf{y}\ \ \bigwedge_{\alpha\in A} E_\alpha)\\
& && \text{[Distribution of } \forall \text{ over } \wedge\text{, quantifier property]}\\
&\longleftarrow (\forall\,\mathbf{y}(\ \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge (\forall\,\mathbf{y} \bigvee_{\beta\in B} \neg\ E_\beta) \vee (\neg\ \exists\mathbf{y}\ \ \bigwedge_{\alpha\in A} E_\alpha) && \text{[De Morgan]}\\
&\longleftarrow (\forall\,\mathbf{y}(\ \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge \bigvee_{\beta\in B} \forall\,\mathbf{y}\neg\ E_\beta) \vee (\neg\ \exists\mathbf{y}\ \ \bigwedge_{\alpha\in A} E_\alpha)\\
& && \text{[Distribution of } \forall \text{ over } \vee \text{ -- implied by]}\\
&\longleftarrow (\forall\,\mathbf{y}(\ \textstyle\bigwedge_{\alpha\in A} E_\alpha) \wedge \bigvee_{\beta\in B} \neg\ \exists\mathbf{y} E_\beta) \vee (\neg\ \exists\mathbf{y}\ \ \bigwedge_{\alpha\in A} E_\alpha)
\end{aligned}
$$

□

B Basic Tree from (1-3)

The following shows a version of the proof tree which does not utilise the negation extension. Numbers represent clauses: recall that '0' represents built-in predicates.

```
ListIds =  [1,[100056,100043,2,[7,[3,[4,
[10071,10066,5,[10071,10066,10071,10078,100063,10056,10033,0],
6,[10071,10066,10071,10079,100064,10056,10034,0],10094,10026,0,0],4,
[10071,10066,5,[10071,10066,10071,10078,100051,10056,10033,0],6,
[10071,10066,10071,10079,100052,10056,10034,0],10094,10026,0,0],10080,0],
10076,0],8,[10075,3,[4,[10071,10066,5,[10071,10066,10071,10078,100063,10056,10033,0],6,
[10071,10066,10071,10079,100064,10056,10034,0],10094,10026,0,0],4,
[10071,10066,5,[10071,10066,10071,10078,100051,10056,10033,0],6,
[10071,10066,10071,10079,100052,10056,10034,0],10094,10026,0,0],10080,0],0,12,
[10078,10113,0],15,[10079,10113,0],10,[1029,1029,10033,11,
[1029,1033,1033,16,[10135,10135,10135,10110,10135,10140,1034,10135,10135,10145,19,
[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],38,
[10071,10071,1022,0],0,0],10107,10122,0,0],0,0],16,
[10135,10135,10135,10110,10135,10140,1034,10135,10135,10145,19,[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],38,
[10071,10071,1022,0],0,0],10107,10122,0,0],0,0],0],0],0],0],0]].
```

C Extension of Tree from (4-7)

The following tree utilises the extension to include negation explicitly.

```
ListIds = [1,[100056,100043,2,[7,[3,[4,[10071,10066,5,
[10071,10066,10071,10078,100063,10056,10033,0],6,
[10071,10066,10071,10079,100064,10056,10034,0],10094,10026,-101,0],4,
[10071,10066,5,[10071,10066,10071,10078,100051,10056,10033,0],6,
[10071,10066,10071,10079,100052,10056,10034,0],10094,10026,-101,0],10080,0],
10076,0],8,[10075,3,[4,[10071,10066,5,
[10071,10066,10071,10078,100063,10056,10033,0],6,
[10071,10066,10071,10079,100064,10056,10034,0],10094,10026,-101,0],4,
[10071,10066,5,[10071,10066,10071,10078,100051,10056,10033,0],6,
[10071,10066,10071,10079,100052,10056,10034,0],10094,10026,-101,0],10080,0],
-30,[[-31,[[-10084]],-32,[[10085,10085,10101,-1042]]]],12,[10078,10113,0],15,
[10079,10113,0],10,[1029,1029,10033,11,[1029,1033,1033,16,
[10135,10135,10135,10110,10135,10140,1034,10135,10135,10145,19,
[-29,[[10093,10009,10090,10012,10090,10012,0,-1032,10090,-10009,10090,-10009,10093,-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,[[1012,10123,10123,1017,10071,10071,-1025]],-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,[10135,10135,10135,10110,10135,10140,1034,10135,10135,10145,19,
[-29,[[10093,10009,10090,10012,10090,10012,0,-1032,10090,-10009,10090,-10009,10093,-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[[1012,10123,10123,1017,10071,10071,-1025]],
-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 (latN(53) is north of latN(58))'.