The Automated Refinement of a Requirements Domain Theory

T. L. McCluskey and M. M. West
School of Computing and Mathematics
The University of Huddersfield, Huddersfield HD1 3DH, UK
t.l.mccluskey@hud.ac.uk, m.m.west@hud.ac.uk

September 16, 1999

Abstract

The specification and management of requirements is widely considered to be one of the most important yet most problematic phases in software engineering. In some applications, such as in safety critical areas or knowledge-based systems, the construction of a requirements domain theory is regarded as an important part of this phase. Building and maintaining such a domain theory, however, requires a large investment and a range of powerful validation and maintenance tools.

The area of 'theory refinement' is concerned with the use of training data to automatically change an existing theory so that it better fits the data. Theory refinement techniques have not been extensively used in applications because of the problems in scaling up their underlying algorithms. In this paper we describe an environment for validating and maintaining a requirements domain theory written in a customised form of many-sorted logic. The environment has been used for several years to maintain a theory (the *CPS*) representing air traffic control separation standards, operating methods and airspace information. In particular, we describe novel theory refinement techniques which we have designed, implemented and integrated within the existing validation environment. These techniques deal with some of the size and expressiveness problems typically encountered when automating the refinement of a large theory. Using a supply of expertly classified training data, we describe the role of a theory refinement tool in the removal of errors and the induction of new parts of the theory from actual logs of separation decisions concerning the flight profiles of air traffic.

1 Introduction

Promoting and maintaining the quality of requirements specifications has a vital role in the engineering of software. Indeed, requirements management is seen as one of the principal problems facing software developers [EC96]. Software projects involving safety-critical elements necessitate that precise, mathematical specifications of their requirements domains be constructed. Such *domain theories*[1] must be validated to satisfy certain major quality objectives such as accuracy and completeness, and during a system's lifecycle the theory is likely to be incrementally updated, and will require re-validation.

A unifying theme in the research areas of knowledge engineering, requirements engineering and formal methods is the construction and validation of requirements represented as formal systems (using rich languages such as RML [GMB94]). In AI, such rigorous approaches to requirements capture have been taken in for example knowledge-based systems [vHF95, MP96, ABvH94] and automated planning [MP97]. Within Software Engineering the use of mathematically precise specifications and methods are still controversial, although it is generally agreed that in a range of applications where the building of a domain theory is feasible, many advantages for the system development accrue. Establishing the basis of a set of requirements in such a precise form supports automated analysis of those requirements to investigate ambiguities, inconsistencies and incompleteness. It also allows one to assess the impact of requirements change in a rigorous manner.

Starting with a domain theory, automation of the software development process has been used in a number of large-scale applications, especially in the safety-critical areas ([LHHR94, HL96]). The advantages of such automation are seen as three-fold [SG96]: as a move towards proof of correctness of implementations, simulation of requirements to support specification development, and automatic generation of efficient implementations.

Given that the production of a precise, abstract domain theory is desirable, a prime concern is its validation and maintenance i.e. ensuring that it is kept accurate and complete. Validation of a domain theory brings with it problems and advantages: it may be harder for a non-computing professional to understand, and may contain many more details than would appear in a conventional requirements document [Par98]. In any case, such a theory can never be considered self-evidently correct, and it must go through a process whereby it is adjusted or refined to be a faithful representation of the domain [EC98, SG96].

On the other hand, the formality brings with it the opportunity for powerful tool support, for validation and maintenance of realistic domain theories is a very time consuming, expensive process where the role of support tools is vital. The validation process is best carried out using diverse techniques, for proof tools alone are not sufficient in discovering the source of inconsistencies [EC97]. For satisfactory validation, the theory needs to be able to stand up to critical

[1] sometimes these are called 'requirements models' to emphasise that they are used to investigate the behaviour of the hypothesised requirements

examination by the customer (or user) [Rus93]. One of the most useful techniques for validation is to *test* an animated form of the requirements [Muk95, WE92]. Even when an animated version is available, however, it is not easy to pinpoint the causes of bugs and subsequently provide the correct revision that eliminates them.

1.1 Theory Refinement

The problem of establishing and maintaining a precise, abstract domain theory can be formulated as one of *theory refinement*[2] (TR). TR is a subfield of Machine Learning concerned with using an expertly classified set of training examples to revise an existing theory [Wro96, RM95]. A theory is viewed as an imperfect representation of a (requirements) domain that needs to undergo refinement to remove errors or to reflect changes in the domain. The example-set is usually acquired from domain experts: they provide input to an animation of the theory in the form of training examples or required behaviours. TR systems typically consist of (a) an animation component, which can apply the theory to training examples, (b) a component which can identify potentially faulty parts of the theory using the results of applying the theory to the examples, and (c) a component which uses TR *operators* to generalise or specialise (parts of) the theory in order to more accurately classify the examples.

TR is related to the established field of Inductive Logic Programming (ILP) [Mug91]; the differences are that an ILP algorithm is given an initial background theory T, some examples E, but no initial revisable theory. The algorithm then has to induce a hypothesis H that together with T explains the examples E. In contrast, TR uses the theory as the initial hypothesis and incrementally changes it until it explains E.

1.2 The IMPRESS project

The work reported here was carried out in a project called IMPRESS. The aim of this project was to develop machine learning techniques and evaluate their application to the validation of requirements domain theories written in customised, many-sorted first order logic. To drive the research we used an air traffic control application where a domain theory had been captured in a previous project called FAROAS [MPN+95]. The theory, called the 'Conflict Prediction Specification' or *CPS*, represents aircraft separation criteria and conflict prediction procedures relating to airspace over the North East Atlantic and is written in many-sorted logic[3] (here called *msl*). The *CPS* had been encased in a tools environment which aids in the automation of the validation and maintenance process. IMPRESS was supported by the UK National Air Traffic Services (NATS), and hence an additional objective was to identify and document errors found in, and

[2] the name derives from its roots in knowledge acquisition. The field is also referred to as *theory patching* or *theory revision*

[3] it is recorded in the 'Formal Methods Europe Applications Database' web site http://www.cs.tcd.ie/FME

refinements carried out on, the *CPS*. Progress towards these objectives and the resultant developments in the original project plan were documented during the project [McC96, McC97b], and the updates of the *CPS* delivered to NATS. Conference papers presented include references [MM98a, MM98b].

1.3 Contribution

The most important contribution of this paper is a method, and algorithms, for applying theory refinement to a large technical theory of functional requirements written in customised many sorted logic. Our achievements can be summarised as follows:

identification of faulty axioms The *CPS* was translated to a logic program so that the *msl* axioms become program clauses and the repair process commenced via the input of training examples. An algorithm is presented which generates and analyses proof trees indicating the success and failure of clauses. In order to cope with the expressiveness demanded by the real application, we extended the meta-interpreter technique for proof tree generation to cover *general logic programs*. This included the problem of the expansion of negated literals, which we tackled via the use of Clark's notion of the completion of a general logic program [Cla78] and via De Morgan's laws.

focusing of repair strategy In order to cope with the scale of the problem, we developed a focused strategy to repairing faulty parts of the *CPS*. We concentrated on repair of numeric components, for given combinations of non-numeric properties. The novel TR algorithm we developed (i) successfully updated boundary values in axioms of the *CPS* to conform to new ATC criteria, and (ii) discovered and corrected a faulty axiom in a manner consistent with expert procedures. The faulty axiom had remained undiscovered during previous validation.

Our results were dependent on certain conditions, the primary ones being that the theory in question must be translatable into an efficient logic program, and must be maintained with a meta-theory containing descriptive information such as the validation status of the axioms. Finally, our experiments confirm that a TR tool should be integrated to, and used in conjunction with, other more 'conventional' domain theory management tools. As well as playing a role in a diverse validation strategy for the theory, they are essential in the validation of new knowledge discovered by TR and to analyse training data for 'noise'.

1.4 Organisation of the Article

The paper is organised as follows. In section 2, we will give a brief introduction to the application domain, and describe the theory that was created to represent it. We briefly describe the tools

environment, and the central tool in the environment, an animator which translates the theory into an executable form, and allows one to 'test' the theory. We start section 3 with a general TR algorithm and demonstrate how it could be applied to the *CPS*. From the failure of the general approach, we construct a more effective algorithm that searches for errors in the 'ordinal' parts of the theory. We describe the blame assignment and theory revisor functions, which input batches of training examples, identify parts of the theory that are most likely to be faulty and output suggestions for refinements to the theory. In section 4, we briefly discuss other processes that have been used to remove errors from the theory. Section 5 discusses related work, and section 6 contains our conclusions.

2 The *CPS* and its Tools Environment

2.1 Domain Description

Air traffic in airspace over the eastern North Atlantic is controlled by air traffic control centres in Shannon, Ireland and Prestwick, Scotland. It is the responsibility of air traffic control officers to ensure that air traffic in this airspace is separated in accordance with minima laid down by the International Civil Aviation Organisation. Central to the air traffic control task are the processes of *conflict prediction* – the detection of potential separation violations between aircraft flight profiles and *conflict resolution* – the planning of new conflict free flight profiles. The controllers have tool assistance available for their tasks in the form of a flight data processing system which maintains detailed information about the controlled airspace, including for example details of all aircraft in an airspace and their proposed flight profiles, organised track systems and weather predictions.

The aim of our initial development work was to formalise and make complete the requirements of the separation standards with respect to the specific task of predicting and explaining separation violations (i.e. conflicts) between aircraft flight profiles, in such a way that those requirements could be rigorously validated and maintained. Each flight through the region has an associated profile, which consists of a sequence of (roughly) five or six 'straight line' segments. The profile also contains other information about the aircraft such as its 'call sign' and its type. Each segment is defined by a pair of 4 dimensional points, and the Mach number (i.e. speed) that the aircraft will attain when occupying the segment. Two different profiles adhere to separation standards if they are either vertically, longitudinally or laterally separated. To cope with the volume of air traffic, controllers draw up an 'organised track system' in advance for each day. This is used by the majority of aircraft and ensures vertical or lateral separation for aircraft on different tracks. Aircraft on the *same* tracks, however, are not therefore vertically or laterally separated, and must be separated longitudinally.

The *CPS* was created to contribute towards the requirements specification for a decision support system for air traffic controllers. Related work in formalisation of air traffic control criteria is

described in reference [DJP97], where a tabular style of specification and a variant of higher order logic is used. Specification of the TCAS family of airborne devices using *Requirements State Machine Language* have been described in references [HL96, LHHR94]. A different approach is taken in [Lyn99], where *hybrid I/O automata* are utilised. Note that the conflict software part of the flight processing system differs from that for collision avoidance. The TCAS collision avoidance systems are aircraft-based and function independently of ground-based Air Traffic Control.

2.2 Producing a Customised Theory of the ATC Domain

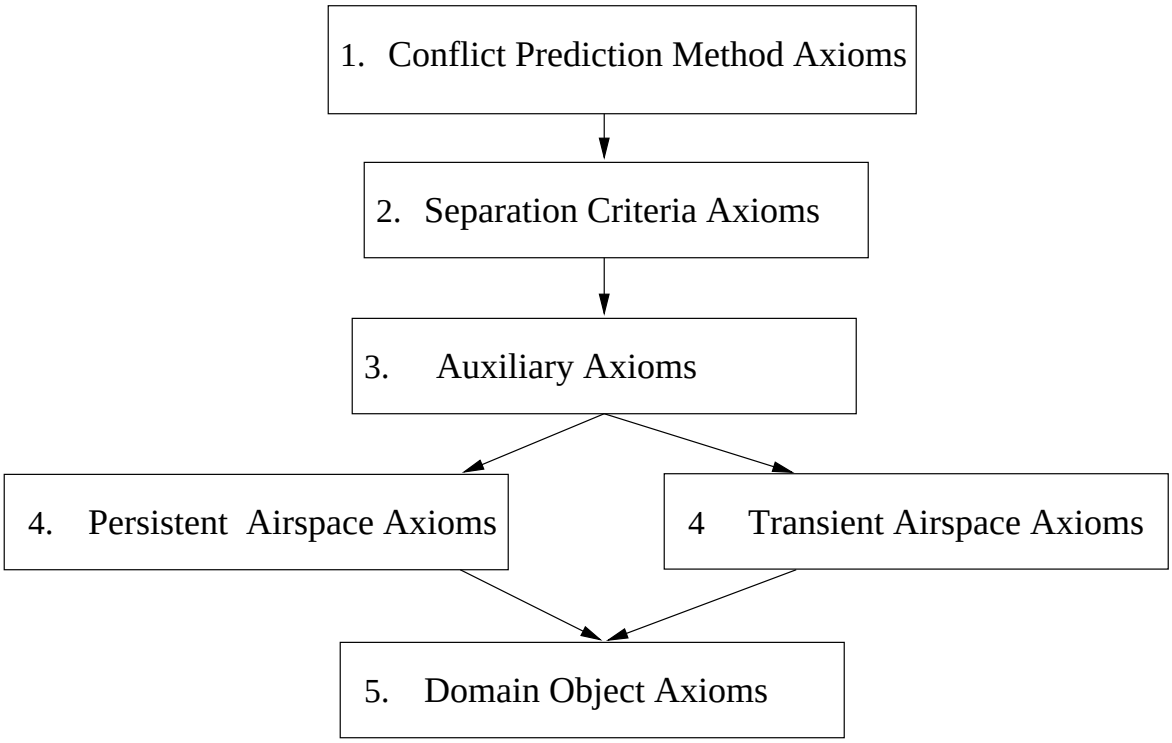

Figure 1: Object Level Axiom Structure in the Domain Theory

An important criteria in developing a large domain theory is to keep the 'semantic gap' between domain and theory as small as possible. This allows the theory's notation, or an equivalent 'pseudo-natural language' form, to be understandable to non-computing professionals (overcoming at least to some extent criticisms of the use of formal systems in requirements analysis [HL96, LHHR94]). We chose many-sorted first order logic to encode the theory for a number of reasons, detailed in [MPN+95]. As msl is a very general language we customised it chiefly through the imaginative and precise use of syntactic constructs. In our ATC theory, all the terminology was chosen to fit in with the source terminology, and our resulting theory is readable and

understandable by air traffic controllers (though they found our lower level object and geometrical axioms - not surprisingly - quite tedious to read).

The theory was constructed on two levels, an object level that reflects the tangible requirements, and a meta-level which includes information about the theory and the language it is written in.

(1) The **object level** consists of a set of *msl* axioms, directly describing the objects, sorts, functions and relations in the domain. The structure of these axioms is shown in Figure 1 - generally the predicates and functions used in the higher levels axioms have their definition at the same level or at a lower level, hence the theory is hierarchical in nature. Axioms are used to define 22 domain sorts which include Aircraft, Latitudes, Longitudes, Flight Levels, Times, Flight Level Ranges, Two, Three and Four-dimensional points, Linear-track points, Segments, Profiles and Airspaces. The axiomatic definition of the sorts are defined in a composite fashion, and are ultimately built on integers, reals, booleans and constant identifiers. For example, a segment (of a profile) is defined as an aggregation of a profile identifier, two four dimensional points, and a speed on the Mach scale.

A typical instance of the domain theory contains over 2000 axioms comprising some 300 non-atomic axioms (levels 1,2,3, and 5 in the figure), 200 atomic axioms containing persistent airspace information, and the remaining axioms containing transient aircraft and aircraft profile data (level 4 in the figure). The non-atomic axioms are fairly complex, containing approximately 2000 predicate instances within them in total. Axioms are first order but otherwise unrestricted logically[4] in the sense that variables from sorts can be universally quantified and existentially quantified, negation and the usual logical connectives can be used and nested to an arbitrary depth, and terms may contain function symbols to an arbitrary depth. When we refer to the *CPS* in the remainder of this paper we mean the axioms in levels 1,2,3,5 and the part of level 4 containing the persistent airspace information (i.e. the *CPS* is distinct from its meta-theory and the transient airspace axioms).

An axiom from the Auxiliary set (Example 1) is given in Figure 2 with its English paraphrase. It defines a temporal relation between two aircraft which at some point are using the same profile track. Variables are universally quantified by default, and represented by capitalised identifiers, whereas the symbol 'E' is used for existential quantification. Example 2 in Figure 3 is an atomic axiom, which is taken from the transient aircraft data, that defines a segment of a profile associated with an aircraft with callsign 'AAL139' to fly at Mach 0.8 at 39,000 feet.

(2) The **meta-theory** is composed of (a) the precise syntactic specification of all the phrases and atoms in the theory (b) a mapping between the phrases and atoms and their natural language equivalent (c) a *validity* level of each axiom. Parts (b) and (c) of the meta-theory are discussed in later sections. Part (a) is the main component and consists of a set of grammar rules (written in Prolog's grammar rule form) containing the syntactic specifications to do with constants, variables, functions and predicates, as well as the general phrase syntax of the customised *msl* language.

[4] although there are some representational restrictions to do with the executable form discussed later

```
"(Segment1 and Segment2 are_after_a_common_pt_
from_which_profile_tracks_are_same_thereafter)
   => [ (the_aircraft_on Segment1
        precedes_the_aircraft_on Segment2) <=>
 E Segment3  [(Segment3
 belongs_to the_Profile_containing(Segment2)) &
        the_entry_2D_pt_of(Segment3) =
              the_entry_2D_pt_of(Segment1) &
        the_exit_2D_pt_of(Segment3) =
              the_exit_2D_pt_of(Segment1) &
    (the_entry_Time_of(Segment3)
is_later_than the_entry_Time_of(Segment1)) ]] "

Example 1

‘For any two segments Segment1 and Segment2,
in the case where Segment1 and Segment2 occur
after a common point from which the tracks of
their profiles are the same,
we say that the aircraft1 on Segment1 precedes
      the aircraft2 on Segment2
if and only if
    there exists a Segment3 in the Profile
               containing Segment2
  such that
    Segment1 and Segment3 have the
       same entry and exit points,
    and
    aircraft2 enters Segment3 later than
       aircraft1 enters Segment1.’

Example 1 paraphrased in structured English
```

Figure 2: An axiom from the Auxiliary Set

The grammar is *two level*: the top level is theory-independent, defining the logical connectives, whereas the lower level contains the concrete syntax which customises the theory. Example 3 defines the syntax of the mixfix predicate used in Example 1, where Segment1 and Segment2 are correctly formed terms of the sort ‘segment’, defined in another part of the grammar. Text appearing literally in the theory appears in square brackets. Each legal form of each sort is also enumerated here, for example the syntax rule that defines Segment (which was used in the Example 2) as an aggregate of other classes is shown in Example 4.

```
"(the_Segment(profile_AAL139_1,
 59 N ; 010 W ; FL 390 ; FL 390 ; 11 37 GMT day 0,
 61 N ; 020 W ; FL 390 ; FL 390 ; 12 26 GMT day 0,
              0.80) belongs_to profile_AAL139_1)"

Example 2

atomic_formula(the_aircraft_on_segment1_precedes
_the_aircraft_on_segment2(Segment1,Segment2)) -->
['the_aircraft_on'], term('Segment',Segment1),
['precedes_the_aircraft_on'],
              term('Segment',Segment2), !.

Example 3

term('Segment',the_Segment(Profile,FourD_pt1,
        FourD_pt2,    Val))  -->
 ['the_Segment'], ['('], term('Profile',Profile),
   [','],     term('4D_pt',FourD_pt1), [','],
 term('4D_pt',    FourD_pt2),[','], val_term(Val),
  [')'], !.

Example 4
```

Figure 3: An Atomic Axiom and some Grammar Rules

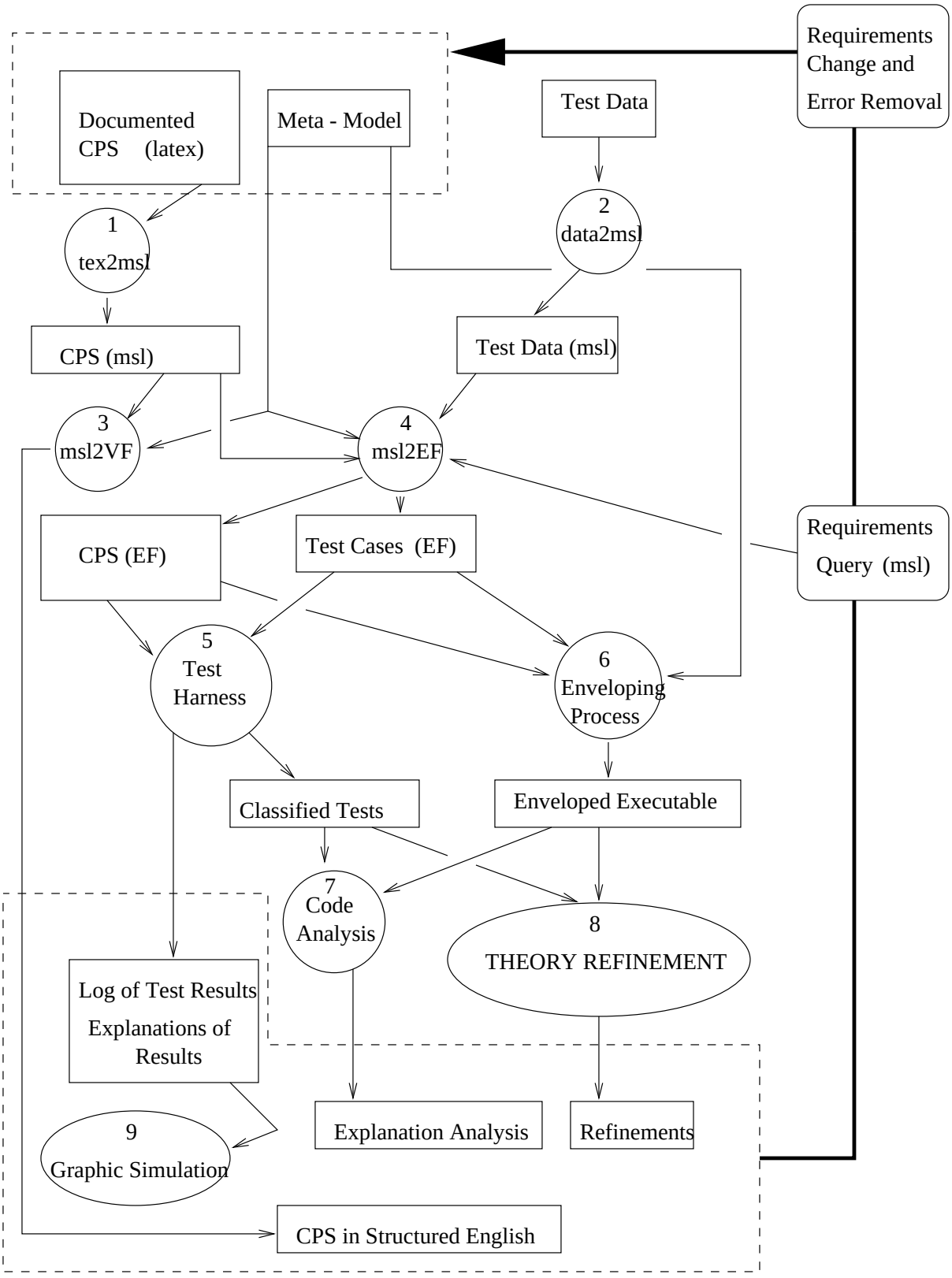

Figure 4: The *CPS* Tools Environment.

2.3 Animating and Testing the Requirements Theory

2.3.1 The *CPS* Tools Environment

Validating and maintaining a requirements domain theory is a complex and a repetitive task, and so automated tools to assist the process are considered essential. An architectural view of the *CPS*'s tools environment is shown in Figure 4. The validation method we advocate is iterative - the inputs to the validation tools are source documents containing the theory (shown in the upper dashed box of the Figure). Additionally, 'training data' in the form of test data, and specific queries and properties, expressed as theorems, are required.

Process 1 (tex2msl) translates the source documentation into a stream of formulae in *msl*, whereas Process 2 translates raw aircraft profile logs obtained from NATS into *msl*, forming the transient aircraft and airspace information (Example 2 is an example of an axiom automatically generated from the raw data). In the ATC application, a batch of test data represents the historic data for several hundred cleared aircraft profiles describing flight plans across the Atlantic in one calendar day. Process 3 translates each axiom into its structured English equivalent, and can be used for visual inspection of the theory. The Prolog grammar described above forms part of tools *msl*2*EF* and *msl*2*VF*, as the Prolog interpreter animates it to form a parser for the theory. Processes 4-8 comprise the animation and theory refinement parts, and will be discussed in detail below. Process 9 is a means whereby aircraft can be viewed 'in flight' moving along their profiles tracks, and is useful for visualising the textual profiles and the pairs of segments deemed in conflict by the *CPS* [McC97a].

The outputs of the validation processes are analysis reports, and suggestions for revisions. Revisions to the *CPS* are carried out tentatively, and will be implemented if the revised *CPS* is passed through all the diverse validation processes (grammatical, manual, batch testing etc) and no errors are found in, or resulting from the revision. Revisions are carried out by changing one or more of the *CPS*'s formulae, the components of the meta-theory, and their associated documentation.

2.3.2 The Animator

Process 3 labelled *msl2EF* in Figure 4 is vitally important as it produces a faithful *operational* form of the *CPS* (*EF* in *msl2EF* means 'execution-form') which is used in the testing and theory refinement processes discussed below. *EF* is a clausal form of *msl* that is executable by a Prolog interpreter, although the structure of the original axioms are preserved, meaning that clause bodies in *EF* may contain compound forms of disjunction and negation. A file of *msl* enters *msl2EF* as a stream of formulae, and each formula is checked sequentially against the stringent syntactic definition residing in the meta-theory. If the syntactic checking process passes without error, then the translator continues by translating the formula into *EF*, respecting the syntactic conventions of Prolog. The translator ensures that each clause in *EF* logically follows from the original

axiom in the *CPS* (cf the transformations of reference [Llo87]). When the *CPS* is translated the resulting set of clauses is called the CPS_{EF}.

The initial capture of the *CPS* was performed without any precondition or restrictions in mind with respect to its operationality (as explained on page 65 in reference [MPN+95]). Although formulae written in the customised *msl* are not restricted logically, therefore, after the *msl2EF* tool was constructed some restrictions emerged to do with translation convention, and the composition of sorts:

(1) A formula must translate to clause(s) containing at least one positive literal. Where a formula contains an '$\Leftrightarrow$', it is assumed to be definitional in its left hand side. In this case the '$\Leftrightarrow$', will be translated to a '$\Leftarrow$'. If a formula's clausal form contains more that one positive literal, the *left most* positive literal will be chosen as the head of the resulting clause. The other positive literals will appear in the clause's body in negated form.

(2) Existentially quantified variables will be operationalised naturally by 'generate and test'. This means that an existential quantifier must be succeeded by a relation or value constructor in the original formula. This form of quantification has to be restricted to appropriate sorts, so that when the output clause is executed, objects of the variable's sort are systematically generated.

(3) Inevitably the translation process results in negated expressions $\neg$ *Exp* within the bodies of some clauses. It must always be the case, however, (and was the case with the *CPS*) that *either Exp* is fully instantiated, or any free variables in *Exp* are existentially quantified. This avoids problems with 'floundering' [Cha88], which can result from the existence of expressions such as $\neg\ \forall xE(x)$.

Function definitions are translated into relations by the creation of matching predicates with an extra slot. Similarly, nested function applications in predicate arguments are 'unpacked' automatically into extra predicates which return intermediate values. The operational form of the auxiliary axiom given in Example 1, that was generated by *msl2EF*, illustrates some of these points, and is shown in Figure 5.

Note the generation of intermediate variables to deal with the lack of function evaluation in Prolog. Functions such as 'the_entry_2D_pt_of' translate into predicates, and values of the existentially quantified 'Segment3' are generated by the Prolog interpreter from all those segments satisfying the 'belongs_to' relation.

2.3.3 The Batch Testing Process

Raw, historical test data is translated by *data2msl* into *msl*, then translated into EF by *msl2EF*, and input to the Test Harness (as shown in Figure 4) as a series of data sets, each set representing the characteristics of mutually-cleared flight profiles passing through 'Shanwick' airspace. This assumes that the order of the input of profiles reflects the order in which they were cleared by

```
the_aircraft_on_segment1_precedes_the_aircraft
             _on_segment2(Segment1,Segment2):-
are_after_a_common_pt_from_which_profile_tracks
_are_same_thereafter(Segment1,Segment2),
    the_Profile_containing(Segment2,Profile1),
    Segment3 belongs_to Profile1,
    the_entry_2D_pt_of(Segment3,Two_D_pt1),
    the_entry_2D_pt_of(Segment1,Two_D_pt2),
    same_2D_pt(Two_D_pt1,Two_D_pt2),
    the_exit_2D_pt_of(Segment3,Two_D_pt3),
    the_exit_2D_pt_of(Segment1,Two_D_pt4),
    same_2D_pt(Two_D_pt3,Two_D_pt4),
    the_entry_Time_of(Segment3,Time1),
    the_entry_Time_of(Segment1,Time2),
    Time1 is_later_than Time2, !.
```

Figure 5: Example 1 in EF

air traffic control officers. The Test Harness executes the definition of main relation (called the *conflict relation*) repeatedly, systematically checking pairs of profiles:

'profiles_are_in_oceanic_conflict(P1,P2,S1,S2)'

Each profile P1 is compared with all[5] other profiles, P2. For given P1 and P2, if this relation succeeds when executed by the Prolog interpreter, then there are at least two segments (S1 from P1, and S2 from P2) that are in conflict, in which case P1 and P2 are deemed to be in conflict. If this relation fails, then the P1 and P2 are separated to the required standard, according to the *CPS*. Mismatches between the expected result and the result returned by the CPS_{EF} drive the refinement processes, and in particular theory refinement.

Our testing process was biased by the fact that we could obtain virtually limitless negative examples of the main conflict relation using records of actual clearances of flight profiles over the Atlantic. On the other hand, we could only obtain a handful of positive examples that had to be specially constructed by air traffic experts.

[5] in fact, for a given P1, we limit this to the *N* chronologically previous profiles cleared before P1. If N = 20, and there were 500 profiles to clear in a day's worth of data, then this would give slightly less than 10,000 runs of the main relation

2.3.4 Executing Specific Queries

A set of queries (which are 'distinguished tests') can be built up over the lifetime of the theory. A test query is a pair $(Query, Result)$ where *Query* is written as a formula in *msl*, and its *Result* is either 'True' or 'False' depending on whether the Query is deemed true or not according to ATC requirements. An example test query is:

((47 N ; 008 W is_on_the_Shanwick_OCA_boundary), True)

After the EF of a Query has been executed, its result is classified as in Figure 6.

Pair:	CPS Result:	Classification:
(Query, True)	True	Truly Positive (TP)
(Query, True)	False	Falsely Negative (FN)
(Query, False)	True	Falsely Positive (FP)
(Query, False)	False	Truly Negative (TN)

Figure 6: Terminology for Classification of Test Results

Queries may consist of calls to the conflict relation, or of any other lower level predicates or functions in the theory. In this way test queries can be used to test *any* part of the *CPS*, and the results of these tests are stored ready for input to the code analysis phase (Process 7 which is briefly described in section 4). Batch Testing described above is thus a special case of query execution, with the clausal form of a test query being:

(profiles_are_in_oceanic_conflict(P1,P2,S1,S2), False)

where S1, S2 are existentially quantified and P1 and P2 are profile identifiers.

2.3.5 Effectiveness of the *CPS* Tools Environment

The parsing and translation components are vital tools in checking and maintaining the accuracy and completeness of the theory. Query execution, batch testing and manual inspection of the Structured English files are also useful in the discovery and elimination of errors. The need for an automatic way to identify the errors, and suggest revisions to eliminate those errors, is apparent, however, when working with large theories that need to be updated regularly to reflect a changing set of requirements. It is with this motivation that we attempted to embed machine

learning techniques in the tools environment, and in particular theory refinement (Process 8). In the next section we describe the practical advances we had to make in order to integrate a TR tool within the *CPS*'s support environment.

3 The Theory Refinement Process

3.1 A General Hill-Climbing Algorithm

At the heart of TR is a learning algorithm which is presented with an initial theory T, interpreted as a structured description of a 'concept' C. T may be split disjointly into a *refinable* theory T_r, and a *fixed* or background theory T_f. For example, T_f could contain the (easily validated) basic definitions of selectors for composite objects, or facts deemed to be true by domain experts. The algorithm inputs evidence in the form of a set of training examples E, where each example is labelled as being a positive or negative instance of C. The algorithm generates a *refinement* (where a refinement is a sequence of revisions) $R(T_r)$ of T_r, such that $R(T_r) \cup T_f$ *explains* the examples, E. It is usual for the refined theories to have to satisfy certain syntactic and semantic restrictions, which are referred to as the *bias*. This implicitly determines the hypothesis space - the space of all refined theories.

An appropriate technique in the implementation of TR is 'hill-climbing'. In a typical cycle of a hill-climbing algorithm revisions to the theory are incrementally generated, evaluated and the best applied to change the theory. Figure 7 presents an outline algorithm typical of general hill climbing approaches used in TR systems [Wro96, RM95]. In Step 1 the disjoint theory is input with a set of training examples which have the form of 'test queries' defined above. At the start of the repeat loop in Step 2.1, the theory is applied to classify the examples and the results are collected into the four classes as shown in Figure 6.

In Step 2.3 the primitive components of the theory (termed 'revision points') that are likely to be faulty are identified. In a logical theory, examples that are falsely classified are explained by a faulty proof, and identification of revision points in the theory is made by assigning blame cumulatively to logic clauses used in the faulty proofs' trees. The list of revision points can be ordered using statistical information about how often each clause occurred in a faulty proof; this frequency of occurrence in Step 2.3 is called the revision point's *potential*.

The inner repeat loop of Step 2.4 iterates through all or some of the list - in the case where the aim is to find the *best* refinement of the theory, it may not be worth trying to consider revisions points that occurred in only a few faulty proofs, if the potential change to the theory is less than the current best change.

Step 2.4.2 inside the inner loop is the heart of the TR process. This executes the refinement operators which generate the potential changes to a revision point F. Operators may alter F by

```
1. Input an imperfect theory T (= Tr U Tf),
   and training examples E;
2. REPEAT
2.1  Apply T to E to obtain lists of TN,FN,TP,FP for C;
2.2  Let Sc := accuracy score of T;
2.3  Call a blame assignment procedure which outputs
     a list of revision points RP in Tr, sorted by
     their potential in descending order;
2.4  REPEAT
2.4.1   remove the first point F from RP;
2.4.2   generate a set RS of revisions for F
        using TR operators;
2.4.3   LOOP for each potential refinement R in RS:
2.4.3.1   calculate accuracy score of (Tf U R(Tr));
2.4.3.2   record best refinement found so far (Rmax)
          with accuracy score (Smax)
2.4.3   END LOOP
2.4  UNTIL RP is empty OR potential maximum score of
        of a refinement to the next F in RP < Smax;
2.5  IF Smax > Sc  THEN
2.5.1  Let Tr := Rmax(Tr);
2.5.2  Store revision R
2.5  END IF
2. UNTIL Smax =< Sc
3. Output the sequence of revisions leading to the
   best refinement found
4. END
```

Figure 7: A General Hill-Climbing Algorithm for Theory Refinement

adding, removing or replacing parts of it. There have been various kinds of operators studied in TR [Wro96]. As a simple example, where the theory is a logic program, an algorithm may delete antecedents of a clause *F* in order to *generalise* it, that is to make it more likely to succeed, generally leading to a theory which classifies more examples as positive. Likewise, adding antecedents to *F specialises* it. Note that this may not have the obvious effect on the whole theory describing some concept; *F* may only appear in proof trees via a call to from a negated literal, hence specialising it has the effect of generalising the the theory as a whole, and vice-versa.

Calculating the accuracy score of the theory for each revision involves a third loop (Step 2.4.3). This involves repeatedly executing the theory with the revisions as they were made, and calculating their resultant accuracy with a standard measure such as the following:

$$accuracy\,score = (TN + TP) * 100/(TN + TP + FN + FP)$$

The hill-climbing algorithm terminates outputting a sequence of revisions in Step 3 such that their sequential operation on the initial theory will improve or equal its initial accuracy.

3.1.1 Complexity of the General Hill-Climbing Algorithm

For a large theory, the most expensive steps are 2.1, 2.3 and 2.4.3.1, where the theory is applied to classify all the training examples. Assume that the outer hill climbing loop is run *n* times, there are at most *m* revision points considered for each run, and there are at most *o* refinements for each revision point. Then, in the worst case, the theory could be applied to the set of training examples $n*(m*o+2)$ times. The size of *o* itself depends on the number and complexity of each refinement operator, and without a restrictive bias such operators can return many possible revisions. Although hill climbing strategies are regarded as the best form of TR [Gre95] a strategy which explores the effect of general refinement operators would falter due to the immense combinatorics of the process, and even with fairly restrictive assumptions recent theoretical research has shown that theory refinement is hard in terms of computational complexity [Gre95, AEK98].

3.2 Application of the General Hill Climbing Algorithm

3.2.1 Experimental Set-up

In our early experiments, for Process 8 in Figure 4, we used a hill-climbing algorithm following the outline given in Figure 7. In Step 1 of Figure 7 *T* was the logic program referred to as CPS_{EF} and *E* was the set of the training examples in execution form (E_{EF}). *EF* was chosen rather than *msl* because of its executability and simpler syntactic structure. The fact that one can easily

trace clauses back to *msl* meant that any refinement in *EF* can be applied also to the original theory. On the other hand, there is a consequent loss of readability in the refinements given in *EF* automatically produced by the TR algorithm. As *EF* is interpretable as a logic program, the head of any clause could be considered as an identifier for the concept. In our case the (main) concept is the conflict relation 'profiles_are_in_oceanic_conflict(P1,P2,S1,S2)', where P1 and P2 and two distinct profiles and S1 and S2 are segments contained within P1 and P2 respectively.

As well as inputting $E_{EF} \cup CPS_{EF}$, the algorithm inputs each clause in an "enveloped" form, containing the clause itself, the clause's identification number ("clause-ID"), and a validity level taken from information in the meta-theory (this is performed by Process 6 in Figure 4). The validity level of a clause *F* is either:

– 'shielded', meaning that *F* is to be left unchanged, or

– 'unshielded', meaning that *F* is a candidate for change, or

– 'fixed', meaning that *F* and any clauses used in the definition of literals in the body of *F* are to be left unchanged.

The set of 'refinable' clauses referred to above as T_r is thus split into 'shielded' and 'unshielded', while the fixed clauses make up T_f. Although a shielded clause's meaning may be revised by a change in the definition of one of its body's literals, the clause itself may not be revised. Typically, top level axioms are put on 'shielded' status as their logically structure may be considered correct, but errors are considered to be present in the definition of their auxiliary axioms. On the other hand, low level numeric definitions may be given the status 'fixed' in that their whole definition is considered correct. The validity level of a clause in CPS_{EF} is the same as the axiom from which it originated in the *CPS*, and the training examples are given a default validity level of 'fixed'. Input to the TR algorithm, therefore, for each clause *F* in $CPS_{EF} \cup E_{EF}$, is a fact of the form:

fact(*F*, <clause-ID>, <validity level>).

Applying the theory to the evidence (Step 2.1) involved executing the logic program using the conflict relation with aircraft profile data. Two aircraft profiles, together with an expert decision as to whether they were in conflict, make up one training example.

In Step 2.2 an alternative score to the one given above was used:

$$\textit{accuracy score} = \textit{if } FN > 0 \textit{ then } 0 \textit{ else } TN * 100/(TN + FP)$$

An occurrence of a FN for the conflict predicate was a serious matter for the CPS_{EF}, as it meant it had not recognised that two profiles were in conflict. After the initial validation phases the the CPS_{EF} gave zero FNs, hence we decided that any revision introducing FNs should be avoided, and for potential revisions the number of TPs should remain stable. In Step 2.3 'blame assignment' algorithms were used to analyse those tests which led to misclassified examples. Proof trees of falsely positive tests were generated and stored, using standard meta-interpreters techniques [SS94]. The clauses occurring in the collection of proof trees generated from a set of false positive examples were accumulated and scored as being faulty depending on their frequency of occurrence in the trees. Finally, for Step 2.4.2, we experimented with simple forms of theory revision operators as mentioned above.

3.2.2 Results

The experiments we performed with an implementation of this algorithm ended in failure, the main factor being the sheer size of the CPS_{EF}.

- Firstly, using the naive algorithm above, with representative sizes of $n = 5, m = 20$ and $o = 3$, a run of the general TR algorithm would, at worst, have to apply the theory to all the training examples 310 times. Given that the CPS_{EF} is a non-optimised automatically generated program, with a large set of examples this could potentially take several weeks of CPU time on our current hardware configuration.
- Secondly, the *CPS* is a full first order theory customised to naturally represent ATC knowledge. Thus it is expressive, containing functors to represent data, as well as negation and disjunction, resulting in the CPS_{EF} containing many clauses with negation and disjunction in their bodies. Standard meta-interpreter techniques, being developed for Horn clause programs, were not sufficiently penetrating to collect full proof trees.
- Finally, the standard TR operators we used only superficially refined the CPS_{EF}. Simply deleting literals from clauses, for example, is unlikely to lead to an improved theory if the theory has already undergone some validation mechanism.

To overcome these problems we developed a new TR algorithm that
(a) uses a powerful blame assignment algorithm, dealing effectively with expressive first order theories, in particular those containing negation in their bodies
(b) contains operators that perform focused, composite and constructive revisions, leading to an efficient yet more penetrating algorithm.
We describe these developments in depth in the next sections.

3.3 Blame Assignment in General Logic Programs

Since all our misclassified tests for the conflict relation of the *CPS* turned out to be falsely positive, our concern was chiefly to design an algorithm for assigning blame to refinable clauses which succeed during the proof of a negative example. The algorithm in Figure 8 is an expansion of Step 2.3 in Figure 7. Note it inputs the enveloped theory T_v, as well as the object level theory and the falsely positive training examples. The method used in refining Step 2.3 determines how evidence is collected for identifying faulty clauses, and is thus crucial to the performance of TR. It outputs *RP*, an ordered list of clause identifiers paired with their potential as a revision point, and the clauses themselves.

```
Procedure Blame_Assignment_FP(in: T,Tv,FP  out:RP);
1 let S = [(c,0,F): c is a clause-ID in Tv, F is c's clause];
2 LOOP for each instance e of FP:
   2.1 Call  generate_trees(in: e; out: Ptree,IDtree)
   2.2 replace (c,N,F) in S with (c,N+M,F),
       where c occurs M times in IDtree
2 END LOOP ;
3 RP := sorted(S);
4 END
```

Figure 8: Expansion of 2.3: Blame Assignment for FPs

A proof tree is a hierarchic structure representing a proof generated as a result of a successful application of the theory to a training example. In generating a proof tree the task is to determine a proof of the instance that is a necessary condition of the theory, where the clauses that have the validity level 'fixed' are not included in the proof.

In section 2 we noted the existence of negated expressions derived from $\neg\ \exists xE(x)$. To cope with the expressiveness demanded by the real application, we extended the meta-interpreter technique for proof tree generation to such negated expressions. Thus tree generation does not cease if a negated expression is encountered. Standard meta-interpreters for generating proof trees cannot cope with programs which include negative literals. The algorithm in figure 9 is derived from a standard meta-interpreter, but has been extended to overcome the problem of negated literals. It describes how a tree generator of the form:

`generate_trees(in: Expr, out: Ptree, IDtree)`

with input expression `Expr` outputs a proof tree `PTree` and a tree of clause-IDs `IDTree` which were involved in the proof of `Expr`. In the algorithm the tree data types are represented algebraically:

`Ptree ::= pleaf predicate | pbranch predicate Ptree | Ptree*`

```
IDtree ::= pleaf number | branch number IDtree | IDtree*
```

where leaf, pleaf, branch, pbranch are type constructors. For `IDTree`, we adopted the convention that if a clause $\mathcal{F}$ had ID *n*, then $\neg\ \mathcal{F}$ in a proof tree was denoted by ID $-n$.

`Expr` can take the form of *a positive literal* (Steps 1.1-1.3); *a conjunction of expressions* (Step 1.4); *a disjunction of expressions* (Step 1.5); *a negated expression* (Steps 1.6 – 1.10). Fixed clauses and unit clauses form the base case(s) of the recursion, the leaf nodes (Steps 1.1 and 1.2) and negative leaf node (Step 1.6). In a standard meta-interpreter, negated expressions become leaf nodes of the tree, and processing would not include cases 1.6 – 1.10. Our extension is based on Clark's notion of the completion of a general logic program [Cla78].
Step 1.6: In the first case, the input expression `not(ExprN)` succeeds and `ExprN` resolves with a fixed clause E which then fails. In that case, `not(ExprN)` and its clause ID `-ID` appear as leaf nodes. In the second case the input expression `not(ExprN)` succeeds and `ExprN` fails because it does NOT resolve with any clause. This could be for two reasons (i) the predicate is undefined or (ii) the predicate is defined but only for unit clauses which do not resolve with `ExprN`. In either case a new clause ID `NewID` is generated (and suitable message output) and `not(ExprN), -NewID` appear as leaf nodes in the appropriate trees.
Step 1.7: The input expression `not(ExprN)` succeeds and `ExprN` resolves with 'n' non-unit unshielded clause(s) with bodies `B1, B2, .. ,Bn` and can thus be expressed as a disjunction using Clark's completion:

```
not(ExprN) :- not(B1 v B2 v .. v Bn).
```

Using De Morgan 's laws this becomes a conjunction:

```
not(ExprN) :- (not(B1) & not(B2) & .. & not(Bn)).
```

Step 1.9: The straightforward use of De Morgan is not possible where the input expression is of the form `not(ExprN1 & ExprN2)`. This is because `ExprN1` may share variables with `ExprN2`. Even if it fails, (i.e. `ExprN1` succeeds) its variables may be required to prevent floundering of the conjoined expression `not(ExprN2)`. For this reason, if `ExprN1` succeeds it is computed and its proof tree is added to the tree, together with the proof of `not(ExprN2)`. For further technical details including related work on negation see references [WBM97, WB97].

3.4 Ordinal, Composite Refinement Operators

To design an effective TR algorithm for use with a technical specification such as the *CPS* one must analyse the likely sources of error - in particular those errors that will be left after syntax checking, batch testing and hand validation. In the light of this analysis we can design more

```
procedure generate_trees(in: Expr, out: Ptree, IDtree)
1. CASE
1.1 Expr resolves with unit E with resolvant Expr',
     WHERE fact(E,id,_ ) is in Tv:
    Ptree = (pleaf Expr'); IDtree = (leaf id) ;
1.2 Expr resolves with (E :- BExpr) with resolvant BExpr',
     WHERE fact((E :- BExpr),id, fixed) is in Tv:
    Ptree =  (pleaf Expr'); IDtree = (leaf id) ;
1.3 Expr resolves with (E :- BExpr) with resolvant BExpr',
     WHERE fact((E :- BExpr),id , F) is in Tv and F \= fixed:
    call generate_trees(BExpr', Ptree', IDtree');
    Ptree = (pbranch Expr' Ptree'); IDtree = (branch id IDtree');
1.4 Expr is a conjunction of literals E1&..&En:
    call generate_trees(E1, Ptree1, IDtree1); .. ;
    call generate_trees(En, Ptreen, IDtreen);
    Ptree =  (Ptree1 .. Ptreen); IDtree = (IDtree1 .. IDtreen);
1.5 Expr is a disjunction E1 v E2:
    IF E1 succeeds THEN call generate_trees(E1, Ptree, IDtree)
     ELSE call generate_trees(E2, Ptree, IDtree);
1.6 Expr is of the form not(ExprN):
    IF ExprN unifies with E
      WHERE fact((E :- BExpr),id, fixed) is in Tv THEN
     Ptree = (pleaf not(ExprN)); IDtree = (leaf -id);
    ELSE
     IF there does NOT exist fact(E,id,_) in Tv
        such that ExprN resolves with E THEN
       generate Newid WHERE fact(_, Newid,_) is not in Tv;
       Ptree = (pleaf not(ExprN)); IDtree = (leaf -Newid);
1.7 Expr is of the form not(ExprN)
     WHERE ExprN resolves with 'n' non-unit clause(s)
     (Ej :- Bj) with resolvants Bj' and fact((Ej :- Bj),idj, Fj)
     is in Tv, j = 1..n  and Fj \= fixed:
    call generate_trees(not(Bj'), Ptreej, IDtreej), j = 1..n;
    Ptree =  (Ptree1 .. Ptreen); IDtree = (IDtree1 .. IDtreen);
1.8 Expr is  of the form not(not(ExprM)):
    call generate_trees(ExprM, Ptree, IDtree);
1.9 Expr is of the form not(E1 & E2):
    IF not(E1) succeeds THEN
     generate_trees(not(E1), Ptree, IDtree)
    ELSE
     generate_trees(E1, Ptree1, IDtree1);
     generate_trees(not(E2'), Ptree2, IDtree2)
      WHERE E2'is E2 with variables shared with E1 instantiated;
     Ptree = (Ptree1 Ptree2); IDtree = (IDtree1 IDtree2);
1.10 Expr is a negated disjunction of literals not(E1v..vEn):
    call generate_trees(not(E1), Ptree1, IDtree1); .. ;
    call generate_trees(not(En), Ptreen, IDtreen);
    Ptree =  (Ptree1 .. Ptreen); IDtree = (IDtree1 .. IDtreen);
2. END
```

Figure 9: Generating Proof Trees with General Clauses

effective TR operators for this type of theory. Below we classify the kinds of error that manifest certain parts of a clause, and the kinds of refinements that help to eliminate those errors.

Each sort in an *msl* specification can be characterised as being either totally *ordered* or not depending upon whether a set of binary, transitive, ordering relations $\{\succeq_1, .., \succeq_n\}$ has been defined for that sort. During the initial validation of the *CPS* we found that errors were related to operators occurring in sorts with complex ordering relations. We call the sorts that are unordered *nominal*, and those that are ordered *ordinal*. Examples of nominal sorts in the *CPS* are *Aircraft, Airspace, Segment, Profile*. Examples of ordinal sorts are *Flight Level, Time* and *Latitude*, where primitive order relations are for example 'is above', 'is at or later than', 'is west of'. These relations are primitive in the sense that they are not defined by any simpler relations of their sort - although they are of course defined themselves in terms of numerical relations. Because of this dichotomy, each clause in the CPS_{EF} has a domain which is a product of sorts

$$X_1 \times \ldots \times X_n \times D_1 \times \ldots \times D_m, n, m \geq 0.$$

where each X_i is an ordinal sort and each D_j is a nominal sort. For example, the clause in Figure 5 has domain of sorts:

$$segment \times segment \times profile \times segment \times two\_d\_pt \times two\_d\_pt$$
$$\times two\_d\_pt \times two\_d\_pt \times time \times time$$

Here n=2, and m=8, *time* being the only ordinal sort. In the current version of the *CPS*'s grammar there are defined 20 primitive order relations, and in the *CPS* itself there are several hundred explicit occurrences of them. Built on these primitives is a set of non-primitive ordinal properties and relations involving ordinal sorts - for example 'one or both of flight levels L1 and L2 are at or below L3' is a relation between 3 flight levels L1,L2 and L3, whereas 'one or both of S1 and S2 are flown at supersonic speed' is a property of two flight segments S1 and S2 which is defined in terms of lower level ordinal relations. Given the number and complexity of ordinal operators, and the likelihood of errors involving them, it was decided to focus a TR algorithm on the revision of clauses containing and defining them.

3.4.1 Analysis of Errors

Consider the investigation of one revision point (clause *F*) during a run of a TR algorithm. Further, assume it contains a set of ordinal, primitive and/or non-primitive operators. Then, if there is an error with one of the operators, say $\succeq$, we have 3 possibilities:

- *Case 1:* $\succeq$ may be incorrectly defined, i.e. there may be an error in its defining clauses
- *Case 2:* $\succeq$ may be correctly defined, but it may not be the correct ordinal operator to be used within *F*

- *Case 3:* $\succeq$ may be the correct ordinal operator to be used within F, but it may contain incorrect parameters.

For Case 1, where $\succeq$ is incorrectly defined (in which case $\succeq$ is a *non-primitive* ordinal operator) then we can assume that the blame assignment algorithm will identify its defining clauses as revision points and it will be refined accordingly.

For Case 2, where it is correctly defined, but it is not what is required in F, then the operator and/or its parameters must be changed. We can deal with the non-primitive case by removing $\succeq$ and replacing it with its definition using an unfolding technique prior to the application of TR operators. Thus if $\succeq$ were the mixfix operator: 'one or both of _ and _ are flown at supersonic speed' applied to two segments $S1$ and $S2$ then it can be replaced by its definition's body

```
( the_machno_Val_on(S1,Val1), Val1 >= 1.00
     or
  the_machno_Val_on(S2,Val2), Val2 >= 1.00 )
```

within the clause F, with parameters within F and the definition unified accordingly. After the clause is expanded its primitive ordinal operators are subject to this same analysis for Case 2 and 3.

In the case where the primitive operator $\succeq$ used is the wrong one, TR may uncover and correct mistakes in the choice of operator within the range of primitive operators available for that ordinal sort. Thus if a sort has a set of binary operators $\{\succeq_1, .., \succeq_n\}$ operator $\succeq$ can be replaced by any distinct $\succeq_i$, for $i \in (1, .., n)$. For example, a slip in the definition of a clause may have resulted in operator 'is_at_or_above' being used rather than 'is_above'. This case is similar to 'conventional' TR operators, that involve deletion, addition or replacement of antecedents from a clause, but used here under the constraint that the refinement involves operators concerned with a specific sort.

3.4.2 Changes to the region of applicability

The most interesting case we found was Case 3, where refinements are needed to clarify requirements involving conditions involving limiting values of the form $x \succeq a$, where a is a constant. These might not have been encoded correctly initially from the expert sources, or they may need to be changed to cope with changing requirements.

Assume we factor out the X_i from the D_j components, for each clause in a logic program. Then for tuples $\mathbf{d}$ satisfying the conditions of the clause there is defined an n dimensional region $\mathcal{R}$ which corresponds to a domain of applicability of the clause. Thus region $\mathcal{R}$ is populated by n-tuples of ordinal variables, where each component variable of $\mathbf{x}$ is ordinal. The region is defined by the

clause's logical expressions $\mathcal{E}(\mathbf{x})$ involving ordinal variables $\mathbf{x}$ and is not necessarily connected. In other words, a clause containing n different ordinal, limiting variables can be thought of as defining an n-dimensional region.

Given a set of positive instances of a clause F, each instance is associated with an n-tuple of ordinal variables $\mathbf{x}$ and an m-tuple of nominal variables $\mathbf{d}$. We should expect positive instances with a given value of $\mathbf{d}$ to have $\mathbf{x} \in \mathcal{R}$. In a similar manner a clause F which does *not* succeed and which is involved in a failed proof tree (or trace) of a negative instance will have $\mathbf{x} \notin \mathcal{R}$. In order for instances to *fail* where they previously succeeded, (and vice-versa), then region $\mathcal{R}$ can be revised to become region $\mathcal{R}'$.

Specialisation of clause F involves adding appropriate constraints to the existing ordinal operators. Consider a set of instances of F, e^{FP}, taken from proof trees of falsely positive training examples. In order that F should fail, we need to *revise* $\mathcal{R}$ to $\mathcal{R}'$ by specialising F, where ideally all the falsely positive training examples are re-classified.

Recalling that the ordinal variables of F are $\mathbf{x} = x_1 \ldots x_n$, we denote the minimum and maximum values of variable component x_i, in the range of instances of e^{FP}, by min_i^{FP} and max_i^{FP} respectively. We induce the following: for every instance of F in e^{FP} to fail, the new specialised region is $\mathcal{R}$ less an n dimensional interval $\mathcal{R}_{FP}$ bounded by min_i^{FP}, max_i^{FP}. We have

$$\mathcal{R}_{FP} = \{(x_1 \ldots x_n) \mid min_1^{FP} \succeq x_1 \succeq max_1^{FP} \wedge \ldots \wedge min_n^{FP} \succeq x_n \succeq max_n^{FP}\}$$

and thus $\mathcal{R}' = (\mathcal{R} \setminus \mathcal{R}_{FP})$ is a sufficient condition for the elimination of the FP instances. In order to accomplish this, we can specialise the clause F as follows: occurrences in the unrevised body of F of the logical expression $\mathcal{E}(\mathbf{x})$ should be replaced in the revised body of F by $\mathcal{E}'(x_1 \ldots x_n)$, which is defined:

$$\mathcal{E}(x_1 \ldots x_n) \wedge \neg\, RL$$

where we have

$$RL = (min_1^{FP} \succeq x_1 \succeq max_1^{FP} \wedge \ldots \wedge min_n^{FP} \succeq x_n \succeq max_n^{FP})$$

Note that, in the presence of truly positive instances of F, the minimum and maximum of each ordinal variable may have to be further restricted so that the overall revision does not result in any falsely positive examples.

Generalisation can be explained in a similar manner: in order for instances e^{FN} to succeed, their $\mathbf{x}$ components are added to the region. We can calculate the region in an analogous manner. Further details of this technique are given in the discussion of the algorithm in Figure 11 below.

3.5 An Effective Theory Refinement Algorithm

In this section we describe a TR algorithm embedded with the blame assignment algorithm and refinement operators described in sections 3.3 and 3.4 respectively. The outline design of this algorithm is shown in Figure 10, and the whole process corresponds to that numbered 8 in Figure 4. Steps 1, 2.1 and 2.2 follow the general algorithm given in Figure 7, and likewise, the end of the algorithm (Steps 2.8 and 3) is similar to the general one. We will therefore restrict our discussion below to Steps 2.3 - 2.7.

Step 2.3 executes the blame assignment algorithm as detailed in Figures 8 and 9. Each element F of the list *RP* output from 2.3 is a 'revision triple', with three components, $F.ID$, $F.score$ and $F.clause$. Recall that clause identifiers $F.ID$ in *RP* may be negative or positive, depending on whether $F.clause$ was found within a negated proof tree or not. All clauses referred to in *RP* that contain non-primitive ordinal operators are expanded in Step 2.4 by unfolding as explained in Case 2 above. Step 2.5 then reduces the set of revision triples by removing those that are shielded or do not involve ordinal sorts. In Step 2.6 the theory is applied to the positive training examples and the traces of all the suspect clauses are stored. Thus, in every part of a proof tree of an FP or TP involving an ordinal operator, the values of each of the ordinal parameters are recorded. Step 2.7.1 removes the F with the highest $F.score$ from RP', and marks each occurrence of an ordinal operator in $F.clause$ as a revision point. Step 2.7.2 attempts to find changes to overcome errors of type Case 2 discussed in section 3.4. It creates new versions of $F.clause$ by replacing ordinal operators with alternative operators of that sort, then evaluating the new versions using the traces. The best revision of all combinations of operator changes and revision points is stored as *Rev*1.

In Step 2.7.3, expanded in Figure 11, we operationalise the region idea discussed in section 3.4 into an algorithm for finding a candidate revision. In Step 2 of Figure 11 the variables occurring in the M ordinal operators of $F.clause$ are collected. In Step 3.2 the logical expression *RL* (defined section 3.4) representing the region R is constructed. In Step 3.3 it is added onto the original clause after the first ordinal operator, and is negated depending upon whether $F.ID$ is positive or not. This addition my be enough to reclassify the FPs into TNs. However, there are cases in which the the process must iterate, where that iteration involves further applications of the theory T (Step 3.4) where:

- clauses have disjunction in their bodies. If part of a clause is of the form $(a\ ;\ b)$ where a and b involve ordinal operators then, even it is possible to construct a condition c representing a region that encloses all ordinal variables' values from the FP traces, the clause with new part $(a \wedge \neg\ c;\ b)$ may still erroneously succeed in step 3.4 if the call to b succeeds.
- some of the ordinal variables have not acquired values at the occurrence of the first ordinal operator of the clause. Some ordinal variables may only acquire values part way through the execution of a clause.
- the spread of values of an ordinal variable taken from FP traces is not contiguous (i.e.

```
1. Input theory T, its enveloped form Tv and examples E;
2. REPEAT
2.1  Apply T to E to obtain lists of TN,FN,TP,FP for C;
2.2  Let Sc := accuracy score of T;
2.3  Call Blame_Assignment_FP(in: T,Tv,FP, out: RP);
2.4  For each F in RP, expand F.clause so that it contains
      no non-primitive ordinal operators;
2.5  Let RP' = {F in RP : F.clause is an unshielded
                  clause that contains at least
                  one primitive ordinal relation};
2.6  Apply T to FP U TP, storing the instance traces of
               each clause F.clause identified in RP';
2.7  REPEAT
2.7.1   remove the first F from list RP' and retrieve
        traces of F.clause;
2.7.2   Use traces to evaluate ordinal operator changes
        to F.clause via the replacement of operators with
        others of the same sort; store best revision Rev1;
2.7.3   call Ordinal_Revisions(in:F,T,FP,TP,traces, out: Rev2);
2.7.4   Apply Rev1(T) to E, and Rev2(T) to E, calculate
        the accuracy scores and record the better revision
        of the two (Rmax) with accuracy score (Smax)
2.7  UNTIL RP' is empty OR the potential maximum score of
        of a revision to the next F in RP'  < Smax;
2.8  IF Smax > Sc  THEN
2.8.1  Let Tr := Rmax(Tr);
2.8.2  Store revision Rmax
2.8  END IF
2. UNTIL Smax =< Sc;
3. Output the sequence of revisions leading to the
   best refinement found;
4. END
```

Figure 10: The Ordinal Theory Refinement Algorithm

```
Ordinal_Revisions(in:F,T,FP,TP,traces out: Rev)
1. N=1
2. collect the distinct limiting variables from the M
   ordinal operators in F.clause;
3. REPEAT
3.1  find the Nth ordinal operator in F.clause;
3.2  construct RL, a logical expression specifying the
     region R, using the sets of FP and TPs;
3.3  IF F.ID is positive
     THEN conjoin the negation of RL to the Nth ordinal
          operator in F.clause
     ELSE conjoin RL to the Nth ordinal operator in F.clause
3.4  Apply T to FP U TP, redefining FP and TP, and storing
     the traces of the applications of F.clause;
3.5  let N := N+1
3. UNTIL N=M or F.clause is not in the proof of any FP;
4. Rev = accumulated revision of original F
4. END
```

Figure 11: Expansion of Step 2.7.3

values of the same ordinal variable from TP traces 'break up' the FP values). Although our algorithm in this case finds the minimum and maximum of the largest contiguous region of FP values, the second of the terminating conditions will not be met.

The iteration ends if the revisions to *F.clause* have removed it from any proofs of FPs, or when all the revision points within *F.clause* (i.e. all the ordinal operators) have been considered. Finally, in Figure 10, Step 2.7.4 applies the theory to all the examples to find the better of the two revisions delivered by Steps 2.7.2 and 2.7.3.

3.6 Evaluation of the Ordinal Algorithm

3.6.1 An Example Application

In this section we will take a particular clause and show how it was refined during an experiment which involved the execution of the algorithm in Figure 10. This will serve to illuminate the algorithm, to illustrate how we used the overall method in practice, and to show the potential of theory revision applied to technical theories of this nature.

The experiment was set up as follows. Two years after the initial version of the *CPS* was completed, the criteria governing aircraft vertical separation were changed so that certain aircraft under certain conditions could fly with a reduced minimum separation. The new criteria were

collectively called Reduced Vertical Separation Minima (RVSM). At the time, the current version (number 3) of the *CPS* was therefore out of date. To obtain classified tests we ran a batch of approximately 5000 separated profile pairs taken from post-RVSM aircraft profile data. These runs were simulations of positive clearance decision taken by Air Traffic Control Officers from among 660 flight profiles for April 15th, 1997. The CPS_{EF} output a set FP consisting of 95 falsely positives. The input to the ordinal refinement algorithm included FP together with a set TP of 12 truly positives, i.e. 12 pairs of profiles that were expertly judged to be in conflict, and which the CPS_{EF} had also judged to be in conflict.

The input theory T was the CPS_{EF} derived from version 3 of the *CPS* (i.e. the pre-RVSM version), composed of 520 clauses. The first 7 clauses were deemed shielded (the axioms corresponding to them had been well-validated by manual inspection), 82 clauses were unshielded, and the rest were fixed. The unshielded clauses were in fact the most complex, and in our judgement the most likely to contain errors. The clause numbered '26' in the CPS_{EF} was one of a set which dealt with vertical separation:

```
the_min_vertical_sep_Val_in_feet_required_for(Flight_level1,
 Segment1,Flight_level2,Segment2,2000):-
 are_subject_to_oceanic_cpr(Segment1,Segment2),
( both_are_flown_at_subsonic_speed(Segment1,Segment2),
  one_or_both_are_above(Flight_level1,Flight_level2,fl(290))
  ;
 one_or_both_of_are_flown_at_supersonic_speed(Segment1,Segment2),
 one_or_both_are_at_or_below(Flight_level1,Flight_level2,fl(430))), !.
```

This clause can be paraphrased as 'The minimum vertical separation value in feet required for two flight levels of two segments is 2,000 ft if both segments are subject to oceanic separation criteria, and either both are subsonic and at least one flight level is above 29,000 ft, or both are supersonic and at least one flight level is at or below 43,000 ft' ('fl' is a flight level constructor in the *CPS*). During blame assignment (Step 2.3) this had potential '84', meaning that it appeared in 84 of the 95 faulty proof trees, and was thus high on the list for refinement. This clause contains non-primitive ordinal operators, hence in Step 2.4 it was expanded, the implicit ordinal operators being replaced with their explicit definitions. The clause below is '26' after expansion (note that this and subsequent examples are obtained from TR logs and the variable naming and layout is obtained through the Prolog 'listing' utility):

```
the_min_vertical_sep_Val_in_feet_required_for(A,B,C,D,2000):-
     are_subject_to_oceanic_cpr(B,D),
     (   the_machno_Val_on(B,H),
         H<100,
         the_machno_Val_on(D,G),
         G<100,
         (   A is_above fl(290)
         ;   C is_above fl(290)
```

```
        ), !
    ;   (   the_machno_Val_on(B,F),
            F>=100
        ;   the_machno_Val_on(D,E),
            E>=100
        ), !,
        (   A is_at_or_below fl(430)
        ;   C is_at_or_below fl(430)
        )
    ), !.
```

This clause is selected by Step 2.5 to be in RP' as it contains ordinal operators, and is in the group of unshielded clauses. The domain of its ordinal sorts is:

$$mach\_no \times mach\_no \times flight\_level \times flight\_level \times mach\_no \times mach\_no$$

represented by variables (H, G, A, C, F, E). In Step 2.6 this list of variables is stored, (partially) instantiated for each occurrence of the clause in the proof tree of a falsely positive or a truly positive. In the experiment, clause 26 had the fifth highest potential (84) out of a set RP' of 19 clauses. Let us investigate the cycle in which in Step 2.7.1, $F.ID = 26$.

In Step 2.7.2 the trace information is used to evaluate $F.clause$ with changes to its ordinal operators. Changes are performed in a top-down manner, hence the first ordinal operator *is_above* will be swapped for each of its sort's alternatives, which are, from the *CPS*'s syntax definition:

$$is\_at\_or\_above, is\_at\_or\_below, is\_below, =, \neq$$

In this case, Step 2.7.2 failed to find any revisions that performed significantly better than the original clause. For Step 2.7.3, the algorithm in Figure 11 changes only the first ordinal operator – the output of the algorithm is the revised clause:

```
the_min_vertical_sep_Val_in_feet_
                  required_for(fl(H),A,fl(G),B,2000):-
   are_subject_to_oceanic_cpr(A,B),
   (   the_machno_Val_on(A, F),
       F<100,
       (   (   not__((number(F),number(80),F>=80))
           ;   not__((number(F),number(86),F=<86))
           )
       ;   (   not__(fl(H)is_at_or_above fl(330))
           ;   not__(fl(H)is_at_or_below fl(370))
           )
       ;   not__(fl(G)is_at_or_above fl(330))
       ;   not__(fl(G)is_at_or_below fl(370))
```

```
      ),
      the_machno_Val_on(B, E),
      E<100, ... etc
```

The rest of the clause was semantically unchanged, as the new additions succeeded in capturing the region that reclassified all the FPs. Step 3.2 of Figure 11 had constructed 3 regions for three of the variables, in effect inducing that the 2000ft separation rule was no longer valid for aircraft flying in the 33,000ft - 37,000ft interval, and flying at a speed between mach 0.80 and 0.86. In fact, this corresponds exactly to the flight level range in which the RVSM criteria are in operation. The speed range intervals turned out not to be part of the RVSM criteria, but were not overly restrictive, as mach 0.8 to mach 0.86 is a common range for subsonic aircraft in this airspace. The accuracy of this revision turned out to be over 90 per cent, and although it was not to be highest in the first hill climbing iteration, we eventually used it as the basis for the RVSM update to version 3 of the *CPS*. The highest revision was one that removed a temporal interval from a clause (numbered 10) concerning longitude separation. This interval contained virtually all of the FP set, and its removal from clause 10 caused it to fail for those examples.

As a run of the ordinal TR algorithm produces some spurious revisions (e.g. the revision of clause 10) that nevertheless score well with the test data at hand, a useful method to check on the quality of a revision is to run the algorithm with a set of training examples from a different calendar day and compare the outputs. To check the quality of the RSVM result above, we ran the pre-RSVM CPS_{EF} with 11016 tests taken from the clearance decisions on 847 aircraft flying over the Atlantic on 4th October 1997. We obtained a set FP of 114 falsely positives, and using the ordinal algorithm clause 26 was again revised, as above:

```
the_min_vertical_sep_Val_in_feet_required_for(fl(H),A,fl(G),B,2000):-
   are_subject_to_oceanic_cpr(A,B),
   (   the_machno_Val_on(A, F),
       F<100,
       (   (   not__((number(F),number(79),F>=79))
           ;   not__((number(F),number(86),F=<86))
           )
       ;   (   not__(fl(H)is_at_or_above fl(330))
           ;   not__(fl(H)is_at_or_below fl(370))
           )
       ;   not__(fl(G)is_at_or_above fl(330))
       ;   not__(fl(G)is_at_or_below fl(370))
       ),
       the_machno_Val_on(B, E),
       E<100, ... etc
```

The accuracy score of this result was again over 90 per cent, and its appearance confirmed the quality of the original revision (although the speed range of aircraft profiles gaining clearance using RVSM criteria was slightly wider than those in the previous experiment). This second batch of training data also provided evidence that the revision to clause 10 above was spurious.

The revision this time involved a temporal region that differed from that of the first experiment, and which gave the clause an accuracy score of only 52 per cent.

3.6.2 An Experiment with an Alternative Concept

A further experiment originally reported in [MM98a] shows the use of the ordinal TR algorithm to an alternative concept to that of the the main conflict predicate. Using 667 profiles from 4th January 1996, we obtained 33 FPs and 5037 TNs out of 5070 runs of the conflict predicate. In this experiment, for expediency, we then focused the revision space on the longitudinal separation criteria rather than the whole CPS_{EF}. The concept was selected by studying the output of blame assignment for all the FPs, and the generalised explanation output for individual FPs, using Process 7 in Figure 4. Longitudinal separation values in minutes can be 5,6,7,8,9,10,15,20 or 30, and the CPS contains formalised criteria for all of these. 75 new training instances were generated from proof trees and proof traces in which a longitudinal separation value of 10 minutes was assigned to two aircraft at least one of which is flying at subsonic speed. The training instances included 25 FN and 50 TP, the concept being:

$$\text{the\_basic\_min\_longitudinal\_sep\_Val\_in\_mins\_}\\ \text{required\_for}(\text{Segment1}, \text{Segment2}) = 10$$

The set TP was generated by re-running the day's worth of instances, and identifying those pairs of profiles in vertical conflict that gave a longitudinal separation of 10 minutes, but were *not* in overall conflict according to both air traffic experts and the CPS_{EF} (thus lowering the possibility of noisy data). The FNs for the concept are derived directly from the 33 false positives from the conflict predicate. The TR algorithm, using ordinal operator replacement, returned a new theory with two clauses altered. 74 of the training instances were now truly positive, and only one falsely negative remained. Significantly, one of the clauses (clause 100) that was revised, defining the predicate:

are_after_a_common_pt_from_which_profile_tracks
_are_same_or_diverging_thereafter_and_at_which
_both_aircraft_have_already_reported_by

was identified as an incorrect reading of an ATC Manual, in that its reference to 'the time of the conflict probe' had been wrongly represented. The revision (a change of two ordinal operators) was subsequently used to help create a version of the *CPS*. An important point about this result was that this clause's ID was recorded in the set *RP* only negatively. Using a blame assignment that was not sensitive to clauses occurring in the negative part of a tree would not have uncovered this faulty clause

3.6.3 Complexity of the Ordinal Algorithm

The steps that apply the theory to the whole example set are 2.1, and 2.7.4, where it is applied twice. Assuming that the outer hill climbing loop is run *n* times, and there are at most *m* revision points considered for each run, this gives a worst case complexity of $n * (2 * m + 1)$. If we take into account the applications of the theory to partial sets of examples (e.g. to the positive set as in step 2.3 and 2.6) then the resulting complexity is of the same order as the complexity of the general hill climbing algorithm at $n * (m * (o + 2) + 3)$, where *o* is the maximum number of times the inner loop of Step 2.7 is run. In effect, these parameters had much lower values than might be expected using the general algorithm because of the new algorithm's ordinal bias, the use of trace information to form and evaluate incremental clause changes, and the construction of non-trivial revisions to clauses between re-applications of the theory.

3.6.4 Summary

The experimental runs of the ordinal TR algorithm in Figure 10 bore out the optimistic complexity analysis in section 3.6.3, and typically amounted to overnight runs of processing using a Sun Ultra 1 with 256 MB RAM running Compiled Sicstus Prolog. At the end of the IMPRESS project, the errors we found in the *CPS* which resulted in wrongly classified tests fell into 3 categories (i) lack of RVSM criteria (ii) problems with the 'time of conflict probe' concept (iii) problems with the *CPS*'s assumption that (locally) the earth can be considered flat. The TR tool helped us identify and eliminate (i) and (ii). In both these cases, the subsequent updates to the *CPS* used the clause revised by the ordinal TR algorithm as a basis. (iii) was discovered during the development of the TR tool and so its discovery can be indirectly attributed to it.

4 Conventional Debugging and Maintenance of the Requirements Theory

While we have concentrated in the last sections on a path to automated debugging and maintenance, we will here summarise the opportunities and procedures that involve conventional methods and their synergy with the more advanced techniques. In some cases they uncovered errors which appear in the environment of the *CPS*, rather than in the *CPS* itself. The procedures are as follows:

(a) theory inspection: The theory can be translated to validation form, using tool *msl2VF* shown in Figure 2. The mapping given in the meta-theory, between the customised msl syntax and natural language, is used for this. The VF produced can then be used for visual inspection by ATC experts. This is described in [MPN+95], and experience has shown that this is most successful in the initial validation of the theory, and for removing errors in the top level axioms.

```
" [ (Segment1 and Segment2 are_after_a_common_pt_
from_which_profile_tracks_are_same_thereafter)
   or
(Segment1 and Segment2 are_after_a_common_pt_from
_which_profile_tracks_are_diverging_thereafter)
        ]        =>
  [ (the_preceding_aircraft_on Segment1
      or_on Segment2 is_faster_by Val mach)
           <=>
 [ [ (the_aircraft_on Segment1
  precedes_the_aircraft_on Segment2)  &
  the_machno_Val_on(Segment1) -
       the_machno_Val_on(Segment2) = Val  ]
   or [ (the_aircraft_on Segment2
  precedes_the_aircraft_on Segment1)  &
  the_machno_Val_on(Segment2) -
       the_machno_Val_on(Segment1) = Val ] ] ]".
```

Figure 12: An Axiom Relating Aircraft Speeds

(b) test log inspection: the Test Harness produces a record of each test run, which includes a brief explanation of every profile pair that is in conflict according to the *CPS*. Where the pair are classified as not in conflict by experts, the explanation may provide clues to the faulty parts of the specification. For example, a percentage of 'falsely positives' were found to be so because the separation value was only very marginally being exceeded. This indicated an error in the geometry, and currently we suspect that the *CPS*'s local flat earth assumption is to blame.

(c) graphical inspection: using the batch results, a collection of test queries can then be formulated with the aim of investigating in greater detail the suspect parts. The test log file can then be input to a graphical flight simulation using a multi-media platform which can display the profiles and conflict area, and simulate the planned aircraft flights [McC97a]. This particular tool helped us spot a batch of 'noisy data'. Inspection of several apparent falsely positive profile pairs using the display clearly showed that the flight profiles were in fact in conflict. The error in this case was tracked down to an incorrect set of flight test data supplied to us.

(d) full explanation analysis: For individual test runs where the conflict predicate succeeds, the Code Analysis process can produce a generalised proof tree, using explanation-based generalisation based on the proof tree output from the blame assignment algorithm, as mentioned above. This is useful for inspecting the outcome of false positive queries, although somewhat tedious as even the generalised proof trees can be over 30 pages in length. This did allow us to spot a very subtle error which resulted from the animation process, and occurred in an auxiliary axiom relating the speeds of two aircraft, shown in Figure 12. Inspection of the proof tree for a misclassified instance in the suspected longitudinal separation clauses revealed that 'the_machno_Val_on(Segment)' was returning a float value when animated using the Prolog in-

terpreter. As a consequence a comparison of the form '0.0199999.. = 0.02' was occurring when the clause corresponding to this axiom was executed.

5 Related Work

The machine learning literature contains many examples of both theory refinement and ILP techniques: see for example [BG96, Wro96]. Automated refinement in parallel with validation and verification is advocated in [Cra96] for the repair of knowledge based systems. There is a relationship between the approaches put forward to validate formal specifications of requirements, and research into the validation of knowledge bases and these are reviewed in [MPWB96]. For a more detailed comparison of validation in software engineering and KBS, the reader can consult [VBC95].

Work in the utilisation of formal methods for safety-critical KBS is described in [Don98]. The paper describes the Esprit Safe-KBS project where the Safe-KBS life cycle was developed, supported by the TRIO specification support package. TRIO is a first order temporal logic language: from an instance of a TRIO specification a 'model' can be generated semi-automatically, providing examples of the functional behaviour of the specified system. Using the model one can do different kinds of proofs, or 'history checking', which is essentially the use of training or test cases.

Theory Patching [AEK98] is described as a type of TR in which revisions are made to individual components of the theory. *Open* components of a theory are those which might be flawed, while *closed* components are fixed. The concern of the latter paper is to determine which classes of logical domain theories the theory patching problem is tractable. The theory patching problem is reduced to the problem of *benign* revisions, where a benign revision is one where closure of the revised component does not affect repairability. This removes some theories from consideration as (eventual) repairs. Theory patching compares with our work in that the validity level of clauses can be compared with the open/closed property of components. Similarly, our strategy of focusing on *ordinal revisions* is another way of restricting theory repair.

According to Wrobel's classification in the review [Wro96], our ordinal TR algorithm exhibits first order, multiple-clause, multiple predicate learning which includes negation-by failure literals. This contrast with reference [RM95], where theories are posed as sets of Horn Clauses. In our work, we utilise a blame algorithm which takes into account 'negative trees' and is thus more accurate than one in which negation is automatically shielded. In a similar manner, Wogulis [Wog93a, Wog93b] has developed a system, A3, that can revise first-order function-free theories containing errors within the scope of a negation. A3 finds the set of all single literal assumptions that could be made to prove each incorrectly classified example. *All* succeeding and failing goals in the proof of an example are candidates. Assumptions are then graded according to the number of examples they cover, and the depth of the assumption. (The identification and

grading of candidate assumptions can be compared with our Blame Assignment.) A3 attempts to repair each clause according to how it uses the assumption, either positively or negatively. In contrast to this work, the CPS utilises functions in a fundamental way, and as indicated in Section 3.2.2, there are intractability problems in searches for faulty clauses which did not focus on ordinal clauses.

The *CPS* contains significant numerical components; machine learning in domains containing significant numerical components has previously been accomplished by using neural networks [OS97]. This is because of problems in encoding numeric properties in the logic programming context. In order to cope with numeric literals in a logic programming context, *Constraint Logic Programming* has been utilised. The Constraint Logic Programming frame subsumes the logic programming frame. In [SR96] it is shown that a discriminant induction problem can be transformed into a Constraint Satisfaction Problem, and hence solvable via Constraint Logic Programming. The approach is also pursued in [AF97], where an algorithm, NUM is presented which generates numeric literals. *Usage declarations* are meta-predicates which restrict the form of the numeric literals. For example a particular numeric literal might be restricted an inequality or to a linear relationship. The work is in its early stages and it is not clear how, or if, it would scale up to deal with real-world examples such as the *CPS*.

As far as we are aware our work is the first to apply machine learning techniques to a requirements domain theory, although, as mentioned above, work most related to our own occurs in the field of KBR. Both areas have to adopt strategies to overcome the complexity pitfalls surrounding the use of TR, where theoretical results suggest that no polynomial algorithm exists to perform global optimisation in hill climbing algorithms [Gre95]. In both MOBAL [SMAU94] and KRUST [CS96], refinement techniques have been used in a kind of incremental fashion, whereas our main effort has been directed towards the use of the automated analysis of a batch of proof trees of successful predicates, and in the case of unsuccessful predicates, the analysis of failure traces. In KRUST [PC96], for example, test cases are used one at a time to refine the KBS, in contrast to our focusing procedure, which uses multiple examples and a form of statistical blame assignment. In MOBAL, an interactive environment for knowledge acquisition has been used with a large security rule base. The tool utilises several learning algorithms in concert with an inference engine and a graphical user interface. Experience with MOBAL is consistent with our own in that ML tools work well in the context of a *diverse* tools environment.

6 Conclusions

A method has been described to integrate a theory refinement process within an existing validation environment encasing a requirements domain theory. The feasibility of utilising such a process to help in the identification and removal of errors, and in the general maintenance of the theory has been demonstrated. The method is general to theories which can be translated into an executable clausal form, and effective for those with rich type information where errors are

most likely to occur in ordinal operator conditions. In this respect this paper demonstrates an application of theory revision to a non-trivial application.

The overall goal of the project was both to experiment and evaluate machine learning techniques in error removal, and to remove errors in the *CPS* via the combination of tools in the validation environment. Considering the latter, over the course of the project the error rates were reduced from around 20 per 1000, to between 1 and 2 per 1000 tests. Further, the source of the remaining errors appears to have been identified.

In the process of this work we have uncovered many areas for future work, most importantly the development of more specialised TR operators for requirements theories, and the development of deeper, and more reflective TR operators. The TR operators reported here do not affect the overall structure of the theory, and it is an open question as to whether they can be extended to do so. Our immediate future work will concentrate on the further development of our validation environment, and the development of a generic version for use with other requirements theories stated in their own customised form of *msl*.

Acknowledgements

The IMPRESS project was supported by an EPSRC grant, number GR/K73152. We would like to acknowledge the help of Chris Bryant, who implemented some of the theory refinement tools, Julie Porteous, for her help in the initial stages of IMPRESS, Julia Sondander of NATS, for the supply of aircraft test data, and Chris Taylor, who implemented the animation tool.

References

[ABvH94] M. Aben, J. Balder, and F. van Harmelen. Support for the formalisation and validation of KADS expertise models. Technical Report KADS-II/M2/UvA/DM2.6a/1.0, ESPRIT, 1994.

[AEK98] S. Argamon-Engelson and M. Koppel. Tractability of theory patching. *Journal of Artificial Intelligence Research*, 8:39–65, 1998.

[AF97] S. Anthony and A.M. Frisch. Generating Numerical Literals during Refinement. In N. Lavrac and S. Dzeroski, editors, *Inductive Logic Programming: Proceedings of the 7th International Workshop, ILP-97* , volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 61 – 76. Springer-Verlag, 1997.

[BG96] F. Bergadano and D. Gunetti. *Inductive Logic Programming, From Machine Learning to Software Engineering.* MIT Press, Cambridge, Massachusetts, US, 1996.

[Cha88] D Chan. Constructive negation based on the completed database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 111–125, Cambridge, MA, 1988. MIT Press.

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

[Cra96] S. Craw. Refinement complements verification and validation. *Int. J. Human-Computer Studies*, 44(2):245–256, 1996.

[CS96] L. Carbonara and D. Sleeman. Improving the efficiency of knowledge base refinement. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning (ICML '96), Bari, Italy, July 3-6, 1996*, pages 78 – 86, July 1996.

[DJP97] N A Day, J J Joyce, and G Pelletier. Formalization and analysis of the separation minima for aircraft in the north atlantic: Complete specification and analysis results. In C M Holloway and K J Hayhurst, editors, *Fourth NASA Langley Formal Methods Workshop (LFM '97), NASA Conference Publication 3356*, 1997.

[Don98] G. Dondossola. Formal methods in the development of safety critical knowledge-based components. In Frank van Harmelen, editor, *Proceedings of the KR'98 European Workshop on Validation and Verification of Knowledge- Based Systems*, 1998.

[EC96] S. Easterbrook and J. Callahan. Independent Validation of Specifications: A coordination headache. In *Proceedings of the Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96))*, pages 232– 237. IEEE Computer Science Press, 1996.

[EC97] S. Easterbrook and J. Callahan. Formal Methods for V&V of partial specifications: An experience report. In *Proceedings, Third IEEE International Symposium on Requirements Engineering (RE'97)*, 1997.

[EC98] S. M. Easterbrook and J. R. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 40(3), 1998.

[GMB94] S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering*, pages 135 – 148. IEEE Computer Science Press, 1994.

[Gre95] R Greiner. The complexity of theory revision. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.

[HL96] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363 – 377, 1996.

[LHHR94] N.G. Leveson, M.P.E. Heimdahl, H. Hidreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

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

[Lyn99] Nancy Lynch. High-level modeling and analysis of an air-traffic management system. In *HYBRID SYSTEMS: COMPUTATION AND CONTROL Second International Workshop (March 29-31, 1999) Berg en Dal, The Netherlands*, 1999.

[McC96] T. L. McCluskey. Progress summary and project plan for step 3. Technical Report impress/1/05/1, School of Computing and Mathematics, University of Huddersfield, UK, 1996.

[McC97a] B. A. McCluskey. MAPS: Using multimedia in aircraft profile simulation. Master's thesis, School of Computing and Mathematics, University of Huddersfield, UK, 1997.

[McC97b] T. L. McCluskey. Progress summary and project plan for steps 4 and 5. Technical Report impress/2/05/1, School of Computing and Mathematics, University of Huddersfield, UK, 1997.

[MM98a] T. L. McCluskey and M. M.West. A case study in the use of theory revision in requirements validation. In *Machine Learning: Proceedings of the 15th International Conference Shavlik, J (Ed.), Morgan Kaufmann Publishers*, pages 368–376, 1998.

[MM98b] T. L. McCluskey and M. M.West. Towards the automated debugging and maintenance of logic-based requirements models. In *ASE '98: Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 105 – 114, 1998.

[MP96] P. Meseguer and A.D. Preece. Assessing the role of formal specifications in verification and validation of knowledge-based systems. In S. Bologna and G. Bucci, editors, *Procedings of the Third International Conference on Achieving Quality in Software*, pages 317–328, London, 1996. Chapman & Hall.

[MP97] T. L. McCluskey and J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence*, 95:1–65, 1997.

[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.

[MPWB96] 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.

[Mug91] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

[Muk95] P. Mukherjee. Computer-aided validation of formal specifications. *Software Engineering Journal*, 10(4):133–140, July 1995.

[OS97] D. W. Opitz and J. W. Shavlik. Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209, 1997.

[Par98] D. L. Parnas. 'formal methods' technology transfer will fail. *Journal of Systems and Software*, 40(3):195–198, 1998.

[PC96] G. J. Palmer and S. Craw. The role of test cases in automated knowledge refinement. In *ES96: The Sixteenth Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems*, pages 75–90, Cambridge, England, 1996.

[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.

[Rus93] J Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, December 1993.

[SG96] L.G. Shaw and B.R. Gaines. Requirements acquisition. *Software Engineering Journal*, 11:149–165, 1996.

[SMAU94] E. Sommer, K. Morik, J. M. Andre, and M. Uszynski. What online machine learning can do for knowledge acquisition - a case-study. *Knowledge Acquisition*, 6(4):435–460, 1994.

[SR96] M. Sebag and C. Rouveirol. Constraint inductive logic programming. In L De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 277 – 294. IOS Press, 1996.

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

[VBC95] A.I. Vermesan and T. Bench-Capon. Techniques for the verification and validation of knowledge-based systems: a survey based onthe symbol/knowledge level distinction. *Software Testing, Verification and Reliability*, 5:233–271, 1995.

[vHF95] F. van Harmelen and D. Fensel. Formal Methods in Knowledge Engineering . Technical report, The University of Amsterdam, 1995.

[WB97] M. M. West and C. H. Bryant. Assigning blame to general clausal form theories. Technical Report impress/2/01/1, School of Computing and Mathematics, University of Huddersfield, UK, 1997.

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

[WE92] M. M. West and B.M. Eaglestone. Software Development: Two Approaches to Animation of Z Specifications Using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.

[Wog93a] James Wogulis. Handling negation in first-order theory revision. In F. Bergadano, L. De Raedt, S. Matwin, and S. Muggleton, editors, *Proceedings of the IJCAI '93 Workshop on Inductive Logic Programming*, pages 36–46. Morgan Kauffman, 1993.

[Wog93b] James Lee Wogulis. *An Approach to Repairing and Evaluating First-Order Theories Containing Multiple Concepts and Negation*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA 92717, US, 1993.

[Wro96] S. Wrobel. First order theory revision. In L De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 14–33. IOS Press, 1996.