The Application of Machine Learning Tools to the Validation of an Air Traffic Control Domain Theory

M. M. West
M.M.West@hud.ac.uk

T. L. McCluskey
T.L.McCluskey@hud.ac.uk

Abstract

In this paper we describe a project (*IMPRESS*) in which machine learning (ML) tools were created and utilised for the validation of an Air Traffic Control domain theory written in first order logic. During the project, novel techniques were devised for the automated revision of general clause form theories using training examples. These techniques were combined in an algorithm which focused in on the parts of a theory which involve ordinal sorts, and applied geometrical revision operators to repair faulty component parts. While we illustrate the feasibility of applying ML to this area, we conclude that to be effective it must be focused to the application at hand, and used in mixed-initiative mode within a tools environment. The method is illustrated with experimental results obtained during the project.

Keywords: Validation; Executability; Domain Theory; Machine Learning.

Electronic version of an article published as : International Journal on Artificial Intelligence Tools, Volume 10, No 4 , 2001, 613-637, doi: 10.1142/S0218213001000684

Received: 27 November 2000
Accepted: 29 May 2001

1 Introduction

Validation of requirements theories for large, complex systems, particularly those involving safety-critical elements, poses a problem which is difficult to solve. This is particularly the case if the theory is updated during its lifetime, for it will then require re-validation. The problems in coordination and communication within large software system projects are reported by Easterbrook and Callahan [1]. The report emphasises the importance of requirements analysis and tracking and the effect a good quality specification has on subsequent design and implementation. In another study [2], an analysis was undertaken of 387 software errors uncovered during the development of Voyager and Galileo spacecraft. This analysis shows that errors in identifying or understanding functional or interface requirements frequently lead to safety-related software errors; this is because safety-related functional faults are particularly related to imprecise or unsystematic specifications. Safety-critical software projects frequently necessitate that precise, mathematical models of their requirements domains be constructed. Validation and maintenance of realistic domain theories is a very time consuming, expensive process where the role of support tools is vital.

One of the most useful techniques for validation is to execute the mathematical model of the specification. In general, formal notations are not executable, although there are some directly executable formal notations such as Spill [3] and OBJ [4, 5]. There are conflicting points of view about the advisability of a notation being executable. An opposing view is Hayes and Jones [6], where it is stated that the ability to be able to reason about a specification is an important validation task, and this is difficult with executable languages. However Fuchs [7], makes the case *for* executability in that it is also important to demonstrate the functionality of a formal specification to a customer or user and this can be achieved by exercising the specification code. A further discussion and summary of both points of view is presented by Gravell and Henderson [8].

The issue of executability is examined by Hoare [9], in which three forms of the greatest common divisor (gcd) function are compared. The first version is in first order logic, which is (in theory) 'executable' via a search of all proofs. The first version is 'refined'[1] and translated to the logic programming language Prolog to produce the second version. However the Prolog used was an early form which did not possess negation, and the resulting Prolog program would not terminate. The third version is a functional program and is a 'refined' version of the second – this does terminate. However the ease of understanding of each version is less good than the previous version. In other words, clarity of requirements decreases with executability of the gcd function. The inference of his paper is that clarity should not be sacrificed for executability. Hoare suggests that the high level language chosen for program implementation should be as close as possible to the original specification. He also suggests the use of such an implementation for a rapid check on the adequacy of a specification.

In order to overcome the perceived difficulty of the customer and/or developer in understanding requirements yet retain clarity, one solution is to *animate* a form of the requirements [10, 11], where a tool is created which will translate the requirements theory into a prototype that is executable on a test-set. The draft IEC standard on industrial safety-related systems [12], states that the aim of prototyping/animation is

> "To check the feasibility of implementing the system against the given constraints. To communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings."

The test-set is usually acquired from domain experts and includes an expected result. Where the results of the execution are not as expected, faulty parts or *flaws* of the theory are identified. However, even when an animated version and appropriate tests are available, it is not easy to first pinpoint and then correct flaws in the theory.

[1] in the sense of being made executable

The field of *machine learning* (ML) is an important area of Artificial Intelligence and within it are possible solutions to the validation problems: these solutions form the principle subject of our paper. ML incorporates knowledge acquisition and knowledge refinement, and encompasses both symbolic learning techniques of induction and deduction, as well as the sub-symbolic areas of artificial neural networks and genetic algorithms [13]. In our work we view a precise requirements theory as an imperfect theory of the requirements domain that needs to undergo refinement to remove the faults in the theory or to reflect changes in the domain. At the symbolic level, a great deal of research has been carried out into the acquisition and refinement of *domain theories* [14] using examples or counter examples as the main input to generalisation algorithms. Some workers have concentrated on inducing theories from scratch [15], whereas others assume we have an imperfect theory already and the examples are used for *theory refinement/ theory revision* (TR) [16], where an initial, flawed theory is refined. Theories may be refined or adapted [17, 18], because they are inaccurate, or because they fail some operating criteria, as in the field of explanation-based learning (EBL) [19]. The TR field is also referred to as *theory patching*, and an early example of its use is the MIS system [20].

This paper describes the development and application of ML tools for the validation and maintenance of a formal specification of a real application. The validation problem involved the uncovering and repairing of faulty parts of the requirements theory called the 'Conflict Prediction Specification' (the *CPS*). This represents air traffic control separation standards, operating methods and airspace information for the Atlantic oceanic area, in a customised form of many-sorted logic (here called **msl**). When the *CPS* was initially engineered, a set of 'conventional' validation tools were built including one which animated the **msl** via translation into Prolog syntax.

The work reported here was carried out in a project called IMPRESS[2]. At the end of the project we had developed ML tools and (in particular) a TR tool which were integrated within the conventional validation environment. The tools embody novel theory refinement algorithms dealing with some of the size and expressiveness problems typically encountered when automating the refinement of a large theory. During the process of developing and using the ML tools, enough flaws were uncovered and removed from the *CPS* to significantly reduce the error rate of our tests using the *CPS*'s animated form. Notably, we found that the tools were useful in **automated maintenance**, in that they could induce new parts of the theory from batches of training examples which embodied changes in practice.

The paper structure is as follows. Section 2 describes the *CPS* and its original validation environment. Section 3 describes existing tools and methods followed by a description of ML tools we developed ourselves. The tools described enhanced the validation environment described in Section 2. Section 4 describes a novel TR tool which automatically induces a refined version of the *CPS* from an existing version and some examples. Section 5 describes work done by other researchers which is related to ours and Section 6 presents some conclusions and future work.

2 The *CPS* and its Tools Environment

2.1 Background

The aim of the IMPRESS project was to develop machine learning tools and evaluate their application to the validation of requirements domain theories written in customised, many-sorted first order logic (**msl**). To drive the research we used an air traffic control application where a domain theory had been captured in a previous project called FAROAS [21]. The theory, called the *CPS*, represents aircraft separation criteria and conflict prediction procedures relating to airspace over the North East Atlantic and consisted of axioms in

[2]IMProving the quality of formal REquirements SpecificationS

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 refinements carried out on, the *CPS*. At certain milestones of the project, updates of the *CPS* were delivered to NATS.

Air traffic in airspace over the eastern North Atlantic is controlled by air traffic control (ATC) 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. A flight profile is the detailed, planned specification of an aircraft's flight which consists of a sequence of several 'straight line' segments. 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. 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; the *CPS* was created to contribute towards the requirements specification for a decision support system for air traffic controllers.

The aim of FAROAS had been 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. The FAROAS project consisted of four phases: (i) scoping and domain analysis; (ii) formalism choice and customisation; (iii) domain capture via the chosen formalism and (iv) creation of an environment for validation. The formalism chosen was **msl** and the reasons for its choice included its suitability for the ATC domain and its expressiveness. The choice of formalism took place early in the project and before the *Formal Requirements Engineering Environment* (FREE) was developed. This supported validation of the *CPS* in 5 key ways:

parsing the 'theoretical form' (TF) of the formal specification and identifying syntactic errors;

prototyping: an 'Execution Form' of the specification (the CPS_{EF}) was automatically generated (in the logic programming language Prolog) and batches of expert-derived test cases were used to compare expected and actual results. Tests take the form of two flight plans in conflict violation with one-another, or else separated to the required standard. The raw test data was translated to **msl** via the FREE environment, and hence to Prolog;

viewing: the specification in non-technical form – a 'Validation Form' (VF) of the specification was generated and was used by Air Traffic Control experts for visual inspection;

reasoning about internal consistency: a reduced minimal set of the specification axioms were shown to be consistent;

animation: ATC experts tested the model themselves via a custom-built interface which had been integrated with the Execution Form prototype.

The parsing and translation sub-processes of the FREE tool form process 1 of Figure 1 and these are outlined in the next subsection.

2.2 Parsing and translating (process 1)

Test batches were supplied to us by NATS, where a batch of test data represents the historic data in 'raw' form for several hundred cleared aircraft profiles describing flight plans across the Atlantic in one calendar day. Raw, historical test data is translated into **msl**, then into

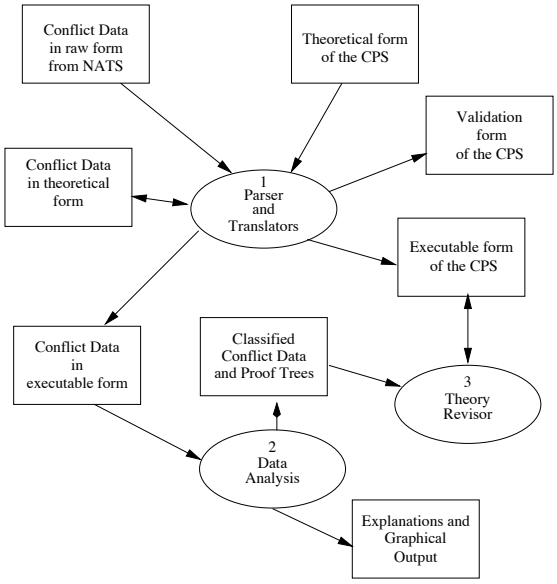

Figure 1: CPS Tools Environment

Prolog. The fragment of the raw test data shown is taken from a batch provided by the ATC and is part of the cleared flights of a single day, the *profile data*. It comprises part of a segmented flight path of a single aircraft *XXXX*[3] flying West at a constant speed of 0.86 Mach and at a height of 35,000 feet. The aircraft is planned to enter the first and second segments at 1042hrs and 1122hrs respectively. *BZZZ* represents the type of aircraft.

```
150497  1042   XXXX   BZZZ  W    M086  M086 350 350

      57/010       57/020       58/030
      1042         1122         1203
      350          350          350
```

The following shows part of the translation to **msl** axioms:

```
"(the_Segment(profile_XXXX, 57 N ; 010 W ; FL 350 ; FL 350 ;
  10 42 GMT day 0, 57 N ; 020 W ; FL 350 ; FL 350 ;
  11 22 GMT day 0, 0.86) belongs_to profile_XXXX)".

"(the_Segment(profile_XXXX, 57 N ; 020 W ; FL 350 ; FL 350 ;
  11 22 GMT day 0, 58 N ; 030 W ; FL 350 ; FL 350 ;
  12 03 GMT day 0, 0.86) belongs_to profile_XXXX)".
```

which translates to Prolog:

```
the_Segment(profile_XXXX,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(10)),fl_range(fl(350),fl(350))),time(10,42,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
```

[3]This represents the call sign. The real call signs and some other details are undisclosed for confidentiality reasons.

```
fl_range(fl(350),fl(350))),time(11,22,0)),0.86)
                      belongs_to profile_XXXX.

the_Segment(profile_XXXX,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(20)),fl_range(fl(350),fl(350))),time(11,22,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(58),long_W(30)),
fl_range(fl(350),fl(350))),time(12,3,0)),0.86)
                      belongs_to profile_XXXX.
```

where sorts such as `FL` are replaced by Prolog functors such as `fl`. The *CPS* relates to separation of flight profiles and contains several hundred 'high level' and 'auxiliary' axioms which mirror complex definitions and procedures, mainly taken from the Manual of Air Traffic Services, Part 2 [22]. 'Lower level' axioms define domain object classes (i.e. the sorts in **msl**) serving as primitives in the model. The *CPS* is translated to the CPS_{EF}, which consists of a set of Prolog *clauses*, where all or part of each clause represents a *CPS* axiom. The highest level axiom of the *CPS* specified the criteria for two different aircraft to be in conflict, that a segment from each of their flight paths be in conflict:

```
"     (Segment1 of Profile1 and Segment2 of Profile2
                  are_in_oceanic_conflict)
      <=>
       [(Segment1 belongs_to Profile1)
        &
        (Segment2 belongs_to Profile2)
        &
        (Segment1 and Segment2 are_in_oceanic_conflict)
       ]  ".
```

which translates to Prolog:

```
profiles_are_in_oceanic_conflict(Segment1,Profile1,Segment2,Profile2):-
    Segment1 belongs_to Profile1,
    Segment2 belongs_to Profile2,
    segments_are_in_oceanic_conflict(Segment1,Segment2).
```

The translation process ensures that each clause in the CPS_{EF} logically follows from the original axiom in the *CPS* following rules established in Lloyd [23]. The process was general enough to cover all the axioms of the *CPS*. The resulting program had some recursion and many negated literals. However it happened to be the case that negated literals $\neg\, p(\mathbf{x})$ (where $\mathbf{x}$ represents the tuple of free variables of p) were of the form that *either* p is fully instantiated, *or* any free variables in p are existentially quantified: $\neg\, \exists\, \mathbf{x}(p(\mathbf{x}))$. Thus problems with 'floundering' were avoided, which can result from the existence of expressions such as $\neg\, \forall\, \mathbf{x}(p(\mathbf{x}))$. This form of negation is sometimes termed *constructive* [24, 23].

3 The Utilisation of Machine Learning

3.1 Existing tools and methods

As explained in the previous section, the IMPRESS project utilised the tools environment and initial specification (*CPS*) inherited from FAROAS. Unfortunately, however, the experience of FAROAS was that these diverse forms of validation, although essential, were still insufficient as far as uncovering and fixing flaws was concerned; for example, tests may succeed for the wrong reasons, and where tests fail (i.e the expert decision is at variance with the prototype's decision) it is still very difficult to identify the faulty or incomplete

requirements. After examining relevant ML techniques, we produced a critical review of the related research and prototype tools [25]. Our review included 'explanation based' tools and methods, tools which 'induced' a theory from examples and general theory revision tools.

As indicated in Section 2.2, the executable form of the *CPS* contains a significant number of literals of the form $\neg(\exists \mathbf{x} p(\mathbf{x}))$. A central requirement for our ML tool development was for a tool which generates a proof tree from a successful Prolog query via explanation-based generalisation (EBG). However the 'text-book' meta-interpreters for generating proof trees [26, 27], cannot cope with general programs (i.e. containing negated literals). In Siqueira and Puget [28], a method is described for generating a failed proof tree, namely, *explanation-based generalisation of failures* (EBGF). A sufficient condition is derived from the failed proof tree which is satisfied by the instance and ensures the failure of the goal. However clause bodies contributing to the failed tree can contain only positive literals. A review and extension is in work by Schrödl [29, 30], where EBGF has been used to aid the generation of trees for proofs which use SLDNF-resolution. For positive literals, a traditional EBG tree is generated. However for negative literals a subsidiary tree is generated via EBGF. The EBGF tree is joined to the main tree via a subsidiary function. The tree generator recurses between EBG and EBGF. The derived formula contains negative goals, disjunctions and existential quantifiers. It is then converted to a set of clauses via standard translation rules [23]. However the requirements for the *CPS* were for a tree which *explicitly* indicated failure nodes and their children as illustrated in Figure 2 of Section 3; the EBG/EBGF method was unsuitable for our purposes.

The *CPS* contains significant numerical components; machine learning in domains containing significant numerical components has previously been accomplished by using neural networks [31]. This is because of problems in encoding numeric properties in the logic programming context. One solution is the utilisation of background knowledge in the framework of Inductive Logic Programming (ILP). For example P-Progol [32], exploits the fact that background knowledge has very few restrictions placed on by the ILP framework. P-Progol utilises background predicates that implement statistical and numerical analysis procedures in order to induce first order theories to scientific problems. Background knowledge available to P-Progol includes (for example) simple arithmetic and trigonometry, inequalities, regression models and numerical differentiation. Two non-trivial case studies are presented, where experimental results are input and constraints on variables which explain the input data are obtained. There are some similarities of the case studies to the *CPS*, in the use of background knowledge in encoding trigonometrical equations and the like. The main differences lie in the underlying model. The two experiments in P-Progol utilise around 10 input ordinal variables whereas the CPS_{EF} domain utilises 22 ordinal variables from the two aircraft. (These are detailed in Section 2 and comprise flight levels, speeds, positions, etc.) Separation criteria also includes many non-ordinal variables such as 'type of aircraft' and whether or not an aircraft meets certain nautical standards.

A further solution to the problem of numeric literals in a logic programming context is the use of *Constraint Logic Programming*. The Constraint Logic Programming frame subsumes the logic programming frame. It has been shown [33], that a discriminant induction problem can be transformed into a Constraint Satisfaction Problem, and hence solvable via Constraint Logic Programming. The approach is pursued by Anthony and Frisch [34], 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 to 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 part of our survey into existing tools we conducted a small feasibility study involving the use of the theory revision tool FORTE [16]. The study concluded that while theory revision or refinement methods as used by FORTE would be useful in validation, the *size* and *expressive* nature of the CPS_{EF} would present various problems. For example,

in order for FORTE to cope with function symbols, the theory requires *flattening* [35], the transformation to an equivalent theory without function symbols. Flattening replaces functors by predicates and in our case this meant that functors representing types were replaced by typing predicates. An example of a simple clause with a functor called `fl_range` is:

```
the_min_Flight_level_of_fl_range(fl_range
    (Flight_level1,Flight_level2),Flight_level2):-
    Flight_level2 is_at_or_below Flight_level1.
```

The same clause is now flattened so that there are no functors. A new predicate `fl_range_pred` is also added to the background theory:

```
the_min_Flight_level_of_fl_range(Flight_level_range,
                                 Flight_level2):-
    fl_range_pred(Flight_level1, Flight_level2,
                  Flight_level_range),
    Flight_level2 is_at_or_below Flight_level1.

/* new 'typing' predicate  */
fl_range_pred(Flight_level1, Flight_level2,
              fl_range(Flight_level1,Flight_level2)).
```

As can be seen, the size of the resulting theory is increased by the introduction of typing predicates (such as `fl_range_pred`). FORTE was run using a subset of the CPS_{EF} and a set of instances. The result was that FORTE over-generalised the theory by removing the typing predicates (such as `fl_range_pred`) and thus 'improved' its accuracy. The incorrect removal of typing predicates could have been prevented by 'shielding' these predicates. However, the pilot study indicated a huge increase of theory size due to the generation of a large number of typing predicates. The typing predicates arise from the existence of the many functors in the CPS_{EF} which model the sorts of the *CPS*, as explained in Section 2.2. The increase in size of an already large specification would have created an intractability problem.

Since the existing tools and methods were not suitable for our purpose, and for reasons of ease of tool integration, it was decided to construct ML tools in-house. The tools developed during IMPRESS are shown in Figure 1 and comprise processes 2 and 3, where process 1 utilises parsing and translation (from the FREE tool). Process 2 classifies and analyses the test output of process 1 (the profile data) and process 3 revises and outputs a new version of the CPS_{EF}.

3.2 Test analysis and validation (process 2)

The profile data is input to a Test Harness as a series of data sets, each set representing the characteristics of mutually-cleared flight profiles passing through 'Oceanic' airspace. This assumes that the order of the input of profiles reflects the order in which they were cleared by air traffic control officers. The Test Harness can be used to execute queries relating to those flight profiles or specifically to check if the *CPS* deems that they are in conflict. One important function of the latter is to repeatedly execute the main relation, concerning conflict prediction, and systematically check pairs of profiles:

$$profiles\_are\_in\_oceanic\_conflict(P1, P2, S1, S2).$$

Each profile P1 is compared with a given number N of chronologically previous profiles P2 cleared before P1. (E.g. for N = 20, and for 500 profiles to clear in a day's worth

of data, around 10,000 runs of the main relation are generated.) 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 and the test result 'positive'. If this relation fails, then the P1 and P2 are separated to the required standard, according to the *CPS* and the test result 'negative'.

Mismatches between the expected result and the result returned by the CPS_{EF} drive the diagnosis of flaws and subsequent repair processes. 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 constructed by air traffic experts. The classification of test results is given in Table 1, where positive and negative test results are compared with positive and negative classification of NATS experts.

Table 1: Terminology for the classification of tests.

		Classification by Theory	
		Positive	Negative
Expert's Classification	Positive	True positive (TP)	False negative (FN)
	Negative	False positive (FP)	True negative (TN)

An example of an FP instance (Example 1) occurs when a comparison is made between profile *XXXX* from Section 2.2 and cleared profile *YYYY*:

```
instance_of_concept(profiles_are_in_oceanic_conflict(
                                the_Segment(profile_XXXX,
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(10)),
      fl_range(fl(350),fl(350))),time(10,42,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
      fl_range(fl(350),fl(350))),time(11,22,0)),0.86),profile_XXXX,

the_Segment(profile_YYYY,
fourD_pt(threeD_pt(twoD_pt(lat_N(56),long_W(10)),
      fl_range(fl(340),fl(340))),time(10,35,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
      fl_range(fl(340),fl(340))),time(11,18,0)),0.86),profile_YYYY)
, fail).

/* Example 1 */
```

As indicated in Figure 1, the instances were utilised to generate explanations of the possible conflicts, including both symbolic and graphical output. They were also used to generate proof trees which provided some indications of 'blame' for faulty clauses of the CPS_{EF}.

3.2.1 Test log and graphical output

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. An example is provided by part of the output from Example 1:

```
 ************ Segments are not Separated ************

CallSign - Type - Mach - non-MNPS? - Steep Sgmt? - Irregular Sgmt?
--------- ------ ------ ---------- -------------- ---------------
yyyy     bzzz   0.86      no          no             no
xxxx     bzzz   0.86      no          no             no

Entry: Time   FL   Lat  Long    Exit: Time  FL   Lat   Long
-----  ----   --   ---  ----    ----- ----  --   ---   ----
yyyy 10 35  340  56 0 10 0        11 18  340  57 0  20 0
xxxx 10 42  350  57 0 10 0        11 22  350  57 0  20 0

Required vertical separation is 2000
Actual separation is 1000
vertical separation contravened

longitudinal separation should be: 15 minutes

The conflict box of xxxx is     60nm lateral 225.0nm longitudinal

yyyy is inside box at distance  50.0nm lateral 98.0nm longitudinal

longitudinal separation should be: 15 minutes

/*Explanation For Example 1 */
```

By examining the output from some of the instances it was found that a percentage of 'false positives' were found to be so because the separation value was only very marginally being exceeded. This indicated a flaw in the geometry, and currently we suspect that the CPS's local flat earth assumption is to blame.

By 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 was 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 [36]. This particular tool helped us spot some invalid data. Inspection of several apparent 'false positives' using the display clearly showed that the flight profiles were in fact in conflict. The invalid data had been sent to us in error.

3.2.2 Proof tree generation

Central to the development of the ML tools was the use of meta-interpreter techniques to re-execute the CPS_{EF} with training examples classified as FP, FN, TN and TP. The meta-interpreter is used to build up proof trees or failure traces in order to pinpoint suspect clauses of the CPS_{EF} by assigning blame to those involved in erroneous proofs or those clauses that failed in a trace that should have produced a proof. In order to store information about clauses in a convenient and efficient manner, each is provided with an automatically generated identity number and the information as to its revisable 'status' (i.e. whether it is revisable or not to be revised). This information is stored in a Prolog structure. The tree generator is derived from a standard meta-interpreter, but has been extended to overcome the problem of negated literals.

Each clause $\mathcal{F}$ in CPS_{EF} was provided with a numeric ID and the tree generator takes as input an expression `Expr` and outputs a proof tree `PTree` and a tree of clause-IDs `IDTree` which were involved in the proof of `Expr`. A problem is that the translation process from **msl** to Prolog resulted in negated expressions `not(Expr)` within the bodies of some clauses. In the case of the CPS_{EF}, negation is such that *either* `Expr` is fully instantiated, *or* any free variables in `Expr` are existentially quantified. A meta-interpreter was written which could

generate proof trees which explicitly represent *negative* rules of the above kind from general logic programs. The meta-interpreter is briefly described using the following small program.

Small Tree Example
The number (ID) of each clause is shown:

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

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


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

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

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 $-n$. In the example program presented above, a call to the tree generator will provide the following response:

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

P = [101,[r(b),[114,t(b),
        -102,[not__(p(b,2)),[112,m(b,3000),0,36000 is
           12*3000,0,not__(36000<24000)]],
        -103,[not__(p(b,3)),[112,m(b,3000),0,36000 is
           12*3000,0,not__(36000<36000)]],
        -104,[not__(p(b,1)),[112,m(b,3000),0,36000 is
           12*3000,0,not__(36000<20000)]]]]],

GenP = [101,[r(X),[114,t(X),
           -102,[not__(p(X,2)),[112,m(X,_B),0,
              _A is 12*_B,0,not__(_A<24000)]],
           -103,[not__(p(X,3)),[112,m(X,_D),0,
              _C is 12*_D,0,not__(_C<36000)]],
           -104,[not__(p(X,1)),[112,m(X,_F),0,
              _E is 12*_F,0,not__(_E<20000)]]]]] ?
```

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

The explicit representation of negative clauses is achieved by first unfolding negative literals and then transforming them using De Morgan's laws. This extension is based on Clark's notion of the completion of a general logic program [37]. The following is an outline; full technical details can be found in the supporting literature [38]. Suppose predicate p is defined by m statements of the form:

$$p(\mathbf{t_i}) \longleftarrow W_i$$

where W_i is a conjunction of literals and $\mathbf{t_i}$ a tuple of variables. The completed definition of p is a disjunction:

$$\forall\,\mathbf{x}(\; p(\mathbf{x}) \longleftrightarrow \bigvee\nolimits_i \exists\,\mathbf{y_i}(\mathbf{x} = \mathbf{t_i}) \;\wedge\; W_i\;)$$

where $\mathbf{y_i}$ are the variables of the original clause. Thus input expressions to the tree generator `not(Expr)` can be expanded into the following conjunction:

$$\neg\ \exists\,\mathbf{x}p(\mathbf{x}) \longleftrightarrow \forall\,\mathbf{x}(\bigwedge_i \neg\,(\exists\,\mathbf{y_i}(\mathbf{x} = \mathbf{x_i})\ \wedge\ W_i)).$$

Since each W_i is a conjunction of literals $\bigwedge_j E_i^j$, in generating the tree we need to obtain proofs of expressions such as $\neg\ \bigwedge_j E_i^j$. Expanding to obtain the disjunction $\bigvee_j(\neg\ E_i^j)$, we may find that (since Prolog uses a left-to-right computation rule) a given component is insufficiently instantiated. In order to make sure that each component is sufficiently instantiated we utilize the following equivalence:

$$\neg\,(L_k) \vee \neg\,(\bigvee_i(E_i^j)) \longleftrightarrow \neg\,(L_k) \vee ((L_k) \wedge \neg\,(\bigvee_i E_i^j)).$$

In the small example, a variable shared by two literals is 'X' which is shared by literals `m(A,X)` and `Y is 12*X`.

Figure 2 contains a pictorial representation of a fragment of a typical tree which was generated during IMPRESS. The tree represents the proof that a pair of aircraft have planned flight paths which potentially violate separation rules. The Figure indicates which nodes succeed and which succeed by failure. The difference between the EBG/EBGF method described in Section 3.1 and our work is that the EBGF tree is defined separately from the EBG tree. In our work the failed predicates are unfolded and integrated with the successful predicates. Thus negation is 'deferred' to the leaf nodes of the tree. This has the advantage that the failed predicates of interest, viz. the unshielded predicates are immediately identifiable. The information obtained from generated proof trees was input

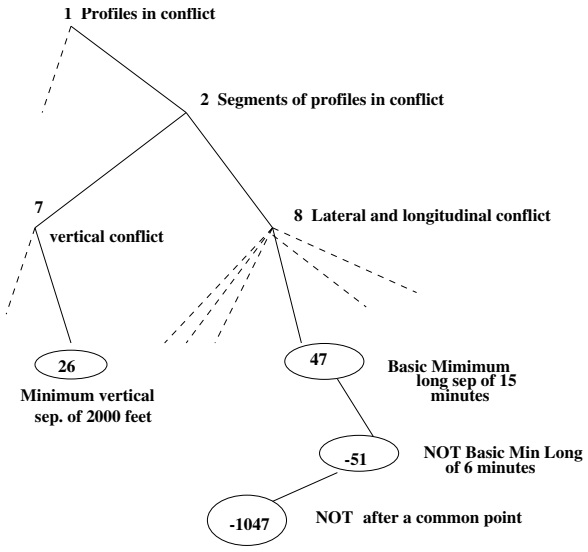

Figure 2: Example of Proof Tree which Includes Negative Literals

into the TR tool, as indicated in Figure 1. A further function of proof tree generation was *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, as mentioned above. This is useful for inspecting the outcome of false positive queries, although somewhat tedious as even the generalised proof trees are 30 - 40 pages in length. This did allow us to spot a very subtle flaw which resulted from the animation process. An auxiliary axiom relating the speeds of two aircraft is shown in Example 2 below:

```
"       [ (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
            ]
          ]
        ] ".
```

```
Example 2
```

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 interpreter. As a consequence a comparison of the form '0.0199999.. = 0.02' was occurring when the clause corresponding to this axiom was executed. This problem resulted from an error in our translation tools rather than an error in the *CPS*.

A further function of Process 2 is a 'blame assignment process'. In order to determine the likely candidate clause(s) for revision, the blame assignment process inputs the theory CPS_{EF} and the false positive examples FP, and outputs a list RP of clauses (which are the revision points) in descending order. In RP each clause and its identification number in CPS_{EF} is associated with a *potential* where the potential of a clause is the number of proof trees it occurs in. (Multiple occurrences of a clause in a single proof are ignored.) Thus clause number -1045 has a potential of 3 if it occurs in 3 proof trees. The next section describes how the output from Process 2 is input to Process 3, viz *Focused Revision*.

4 Focused Revision (Process 3)

The blame assignment and explanation-based tools discussed above focus the user's attention on those parts of the theory that are likely to be faulty. We wanted to go much further in this project: our aim was to construct batch processes to input the results of blame assignment, in the form of an ordered list of faulty clauses, and output the most promising refinement(s) to the clauses. The resulting updated theory would then be submitted to the other validation processes for checking.

To employ a theory refinement approach we have on the one hand to make sure it is effective in suggesting useful changes to the theory, while on the other hand overcome the significant computational complexity problems of the underlying algorithms. Theory refinement algorithms are characterised by the **refinement operators** they embody. These define the primitive changes that are made to the theory by the refinement algorithm. A very general approach to theory refinement is to use coarse grained 'syntactic' refinement operators [39, 16]. These employ techniques such as deleting clauses or deleting/adding

literals from/to clauses. Unfortunately, in our experience with the *CPS* such operators are superficial in their effect, and research prototypes containing them appear to be limited in scope. Simply deleting literals from clauses, for example, is unlikely to lead to an improved theory if the theory has already undergone some validation mechanism.

Fine grained operators which have the potential to refine the internal content and structure of literals would be more effective, but exacerbate the complexity problems. To design an effective TR algorithm for use with a non-trivial 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. Hence we managed the complexity problems by designing what we termed **focused** and **composite** TR operators. These are based around the sort structure of the *CPS*. The CPS_{EF} is automatically generated from the *CPS*, whose sorts can be characterised as being either *totally ordered* or not, depending on the properties of the relations associated with it. During the initial validation of the *CPS* we found that errors were associated with complex ordering relations and their sorts. We call the sorts that are unordered *nominal*, and those that are ordered *ordinal*, where each ordinal sort is associated with a set of binary, transitive, ordering relations $\{\succeq_1, .., \succeq_n\}$.

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* etc. An example of an ordinal atom is:

$fl(330)$ *is_above* $fl(290)$

where $fl(330)$ and $fl(290)$ are of sort *Flight Level*. Therefore, each clause in the CPS_{EF} has a domain which is a product of sorts:

$$X_1 \times \ldots \times X_n \times D_1 \times \ldots \times D_m$$

where each X_i is an ordinal sort and each D_j is a nominal sort. For example the clause providing the conditions for aircraft to have a minimum vertical separation of 2000 feet, where A, C are flight levels and B, D are segments which refer to flight levels A and C respectively is:

```
the_min_vertical_sep_Val_in_feet_required_for(A, B, C, D, 2000) :-
 ( both_are_flown_at_subsonic_speed(B, D),
        ( A is_above fl(290)   ;   C is_above fl(290))
   ;
  one_or_both_of_are_flown_at_supersonic_speed(B, D),
         (A is_at_or_below fl(430) ; C is_at_or_below fl(430)) ).
/* Example 3*/
```

This clause has domain of sorts *Flight_level* × *Segment* × *Flight_level* × *Segment* and 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 are subsonic and at least one flight level is above 29,000 ft, or at least one is supersonic and at least one flight level is at or below 43,000 ft"

The *CPS* contains several hundred ordinal atoms. Most of these are primitive, as in the examples given, in the sense that they are not decomposable into simpler relations. There are also non-primitive ordinal relations - these are ones that are defined immediately in terms of primitive ordinal relations. For example the relation *one_or_both_are_at_or_below* takes 3 flight levels as arguments, and is defined in terms of the primitive relation *is_at_or_below*.

Given the number and complexity of ordinal relations, and the likelihood of errors involving them, it was decided to focus a TR algorithm on the revision of clauses containing and defining them. (The nominal components were left fixed.) In the CPS_{EF} there are many clauses involving limiting values of the form $x \succeq a$, where a is a constant. (See Example 3.) These might not have been encoded correctly initially from the expert sources, or they may need to be changed to cope with changing requirements.

4.1 A refinement operator that changes ordinal regions

As well as focusing in on parts of the theory, refinement operators need to be *composite* in the sense that they compose of a number of single refinements. This increases the efficiency of the TR process. The refinement operator we describe below is both focused and composite in this sense.

Assume we factor out the X_i from the D_j components, for each clause in the CPS_{EF}. Then for tuples 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. Applying the clause in Example 3 to profiles XXXX and YYYY of Example 1 it can be seen that (A,B,C,D) is in the domain of applicability of the clause where:

```
A = fl(350)
B = the_Segment(profile_XXXX,
   fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(10)),
                              fl_range(fl(350),fl(350))),time(10,42,0)),
   fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
                              fl_range(fl(350),fl(350))),
   time(11,22,0)),
   0.86)

C = fl(340)
D = the_Segment(profile_YYYY,
   fourD_pt(threeD_pt(twoD_pt(lat_N(56),long_W(10)),
                               fl_range(fl(340),fl(340))),time(10,35,0)),
   fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
                             fl_range(fl(340),fl(340))),
   time(11,18,0)),
   0.86)

/* Variables from Example 1 */
```

Suppose that a clause $\mathcal{F}$ contains an ordinal variable x, and $\mathcal{R}$ is $a \succeq x \succeq b$. Assume we calculate values c, d of x, $(c \succeq d)$, where c is the maximum value of x found in FP training examples, and d the minimum of the values. Then a useful refinement operator would be to remove the overlapping part of interval $c \succeq x \succeq d$ from $a \succeq x \succeq b$. In order to accomplish this, we can specialise the clause $\mathcal{F}$ as follows: for each ordinal variable x, occurrences in the unrevised body of $\mathcal{F}$ of some logical expression $\mathcal{E}(x)$ should be replaced in the revised body of $\mathcal{F}$ by $\mathcal{E}(x) \wedge \neg (c \succeq x \succeq d)$.

This idea generalises for potentially faulty clauses with several ordinal variables e.g. A,C of Example 3. Note that, in the presence of true positive instances of $\mathcal{F}$, the minimum and maximum of each ordinal variable may have to be further restricted so that the overall revision does not result in any false positive examples. For detailed algorithms implementing our refinement operators the reader is referred to our associated publication [40].

Generalisation can be explained in a similar manner: in order for FN instances to succeed, appropriate intervals are added to the region, and the region is calculated in an analogous manner.

4.2 Testing the composite refinement operator

New requirements for vertical separation of aircraft provided the opportunity for testing the revision operator described above. The new criteria involved new conditions for flight level *intervals* for both aircraft undergoing a conflict probe, for minimum separations of 1000 feet and 2000 feet. The new criteria were collectively called Reduced Vertical Separation Minima (RVSM). Version 3 of the *CPS* (the current version at the time) did not reflect this and was therefore out of date. (See Example 3 for a clause representing part of the old criteria for 2000 feet.) 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 decisions taken by Air Traffic Control Officers from among 660 flight profiles for April 15th, 1997. The CPS_{EF} output a set FP consisting of 82 false positives, the rest being true negatives. The input to the ordinal refinement algorithm included (i) the set FP ; (ii) a set TP of 12 true positives; (iii) a set TN of 12 examples randomly taken from the several thousand negatives in the batch run. The set TP consisted 12 pairs of profiles that were expertly judged to be in conflict, and which the CPS_{EF} had also judged to be in conflict. These positive examples of profile conflict had been supplied to us by air traffic experts. The following trace is taken from the output of the first round of revisions to the CPS_{EF}:

```
the_min_vertical_sep_Val_in_feet_required_for(A, B, C, D, 2000) :-
        (   the_machno_Val_on(B, H),
            H<100,
            limitvar(1),
            (   (   not__(is_greater_eq(H,80))
                ;   not__(is_less_eq(H,86))
                )
            ;   (   not__(is_greater_eq(A,fl(330)))
                ;   not__(is_less_eq(A,fl(370)))
                )
            ;   not__(is_greater_eq(C,fl(330)))
            ;   not__(is_less_eq(C,fl(370)))
            ),
......
26 with accuracy, from REGION, is  93.33333333333333
.......
```

The revision process selects those clauses with limits (ordinal relations of the form $x \succeq a$) which appeared as faulty in the blame assignment stage. These 'limits' are marked in order using a dummy 'limitvar' literal, then revisions are made as explained above. The revision given in Example 3 had constructed altered intervals for flight levels A, C and speed range H. For A and C, it induced that the 2000ft separation rule was no longer valid for aircraft flying in the 33,000ft – 37,000ft interval, and flying at subsonic speed. The region for supersonic speeds was unaltered. In fact, this corresponds exactly to the flight level range in which the RVSM criteria are in operation.

The region change to H, the speed, was made firstly by the algorithm expanding out the *both_are_flown_at_subsonic_speed* predicate in Example 3, as it is made up directly of ordinal relations. Secondly, the speed interval for all the examples which no longer require 2000ft separation was induced to be between mach 0.8 and mach 0.86. This restriction did not represent any change in the theory, but reflected the fact that mach 0.8 to mach 0.86 is a common speed interval for subsonic aircraft in the Atlantic area.

The presence of both true negatives and true positives in the training runs is required to limit the generalisation operators' scope. With only false positives, for example, the TR operator could make arbitrarily generalisations such as lowering separation distances to zero (which would of course change all FPs into true negatives).

4.3 A refinement operator that replaces ordinal relations

As well as errors appearing in limits, a common error is to encode the wrong relation within an ordinal atom. Ordinal relations are grouped by sort in the *CPS*. For example, altitudes have the following associated relations:

$$is\_at\_or\_above, is\_at\_or\_below, is\_above, is\_below, same\_Flight\_level$$

and similarly for time:

$$is\_at\_or\_later\_than, is\_earlier\_than, is\_at\_or\_earlier\_than$$

We designed a refinement operator that in essence 'replaces' one ordinal relation for another in its group. For example, if the rule in Example 3 was identified as being potentially faulty, then each of the occurrences of *is_above* and *is_at_or_below* could be replaced by any of the other three relations in altitude's group. The TR algorithm replaces each ordinal relation with a new relation from its sort, and checks its fitness using trace information. It then implements the most successful relation and evaluates the revised clause's accuracy using a full test run.

4.4 Testing the replacement operator

An experiment was conducted to test the application of the replacement operator in our TR algorithm [41]. To overcome the dearth of positive examples of aircraft profiles in conflict, in this experiment we focused the revision space on the longitudinal separation criterion rather than the whole CPS_{EF}.

Using 667 profiles from 4th January 1996, we obtained 33 FPs and 5037 TNs out of 5070 runs of the conflict predicate. Longitudinal separation was selected by studying the output of blame assignment for all the FPs, and the generalised explanation output for individual FPs. (See for example the proof tree in Figure 2.) From these, we suspected a flaw in the specification of longitudinal separation values. 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 training instances of the longitudinal separation predicate 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:

$$the\_basic\_min\_longitudinal\_sep\_Val\_in\_mins\_required\_for(Segment1, Segment2) = 10$$

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}. This lowered the possibility of noisy data, since as the profiles are all separated but in vertical conflict, the CPS's decision that they are laterally/longitudinally separated is correct, and hence the choice of a particular separation value is more likely to be correct.

The FNs for the concept are derived directly from the 33 false positives from the conflict predicate. The TR algorithm, using ordinal relation replacement, returned a new theory with two clauses altered. 74 of the training instances were now true positive, and only one false negative remained. Significantly, one of the clauses (clause 1047) that was revised, defined the predicate:

$$are\_after\_a\_common\_pt\_from\_which\_profile\_tracks\_are\_same\_or\_diverging\_thereafter\_and\_at\_which\_both\_aircraft\_have\_already\_reported\_by$$

The definition of this predicate was discovered to be 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 relations) was subsequently used to help create a new version of the *CPS*. An important point about this result was that, as can be seen from Figure 2, the ID for this clause was recorded in the revision points only negatively. In other words the negative part of the proof trees had to be explored to uncover the contribution made by this rule to faulty proof trees. 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.

5 Results and Comparison with Related Work

The systems we have produced have been implemented in compiled versions of Prolog, and run on Unix or PC platforms. Approximately 30,000 distinct tests have been processed using the CPS_{EF}, the overwhelming majority via the execution of the conflict relation with negative examples. The results of the tests have been used to maintain the *CPS* during the course of the project resulting in the current version of the *CPS* [42]. During the course of the project, the percentage of FPs during testing has decreased by a factor of 10, to less than half of one percent.

As far as we are aware our work is the first to apply machine learning tools to a requirements domain theory, although related work occurs in the field of Knowledge-based Refinement (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 [43]. In both MOBAL [44] and KRUST[45], 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 [46], 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.

According to Wrobel's classification [39], our ordinal TR algorithm exhibits first order, multiple-clause, multiple predicate learning which includes negation-as-failure literals. This contrast with FORTE [16], 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 [47] 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 3.1 there are intractability problems in searches for faulty clauses which did not focus on ordinal clauses.

6 Conclusions and Further Work

The project has succeeded in applying ML techniques, in particular theory refinement, to the area of validation of formal requirements models. The design, implementation, and testing of integrated tools making up an automated validation environment has been accomplished with techniques incorporating novel forms of blame assignment and theory revision. The TR tool itself has succeeded in revising the theory to take into account the new RVSM rule, and to eliminate an error in longitudinal separation criteria.

The main research contributions of the project are (i) it has shown the feasibility and advantages of integrating machine learning techniques into a validation environment for a formally specified requirements theory and (ii) it demonstrates that when a formal set of requirements have been captured, a largely automated, integrated and diverse set of validation procedures are essential to ensure and maintain the accuracy of the theory.

The project showed that it is feasible to use ML tools in the requirements management phases of a safety-related software project, where the requirements theory is formally represented and can be translated to an executable language. On the other hand, it also illustrated that to be effective these tools need to be: (i) customised to the application at hand e.g. with specific refinement operators; (ii) used in a 'mixed-initiative' mode - that is directed by a human expert; (iii) used in conjunction with other tools within a larger validation environment. The work reported here opens up possibilities for further research. These include the development of more specialised TR operators for requirements theories. 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 creation of a generic tool for use with other requirements theories stated in their own customised form of **msl**.

Acknowledgements

FAROAS was funded directly by NATS, whereas the IMPRESS project was supported by an EPSRC grant, number GR/K73152.

References

[1] 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.

[2] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, 1993. IEEE Computer Society Press.

[3] F Kluzniak and M Milkowska. Spill - a logic language for writing testable requirements specifications. *Science of Computer Programming*, 28((2-3)):193–223, April 1997.

[4] S M Eker, V Stavridou, and J V Tucker. Verification of Synchronous Concurrent Algorithms Using OBJ3: A Case Study of the Pixel Planes Architecture. In G Jones and M Sheeren, editors, *Designing Correct Circuits*, pages 231–252. Springer, 1991.

[5] Gerrard Software. *ObjEx User Reference Manual*. UK, 1988.

[6] I J Hayes and C B Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[7] N E Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, pages 323–334, September 1992.

[8] A Gravell and P Henderson. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11(2):104–110, March 1996.

[9] C A R Hoare. An overview of Some Formal Methods for Program Design. *IEEE Computer*, pages 85 –91, 1987.

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

[11] 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.

[12] Draft IEC 1508. *Functional Safety – Safety-Related Systems (7 parts).* IEC Draft Standard. IEC SC65A (Secretariat) 123, June 1995.

[13] W. Bechtel and A. Abrahamsen. *Connectionism and the Mind; an Introduction to Parallel Processing in Networks.* Blackwell, first edition, 1991.

[14] P. Langley. *Elements of machine learning.* Morgan Kaufman, 1996.

[15] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

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

[17] L. Asker. Improving accuracy of incorrect domain theories. In *Proceedings of the 11th International Conference on Machine Learning: ML'94*, pages 9–27, 1994.

[18] P. M. Murphy and M. J. Pazzani. Revision of Production System Rule-Bases. In *Proceedings of the Eleventh International Workshop on Machine Learning*, 1994.

[19] T. L. McCluskey. Explanation-based Learning. In Z. Ras and M. Zemankova, editors, *Intelligent Systems: State of the Art and Future Directions.* Ellis Horowood, 1990.

[20] Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[21] 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.

[22] 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.

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

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

[25] C. H. Bryant and M. M. West. Machine learning for IMPRESS. Technical Report impress/1/02/1, School of Computing and Mathematics, University of Huddersfield, UK, 1996.

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

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

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

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

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

[31] 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.

[32] A Srinivasan and R. Camacho. Numerical reasoning with an ILP system capable of lazy evaluation and customised search. *Journal of Logic Programming*, 40(3):185–213, 1999.

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

[34] 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 *LNAI*, pages 61 – 76. Springer-Verlag, 1997.

[35] C. Rouveirol. Flattening and saturation: Two representation changes for generalisation. *Machine Learning*, 14(2):219–232, 1994.

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

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

[38] 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.

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

[40] T. L. McCluskey and M. M. West. The automated refinement of a requirements domain theory. *Journal of Automated Software Enginnering, Special Issue on Inductive Programming*, 8(2):195 – 218, 2001.

[41] 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.

[42] T. L. McCluskey. The OACC conflict prediction specification: Version 4. Technical Report impress/3/01/1, School of Computing and Mathematics, University of Huddersfield, UK, 1998.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.