Towards the Automated Debugging and Maintenance of Logic-based Requirements Models

T. L. McCluskey and M. M. West
School of Computing and Mathematics
The University of Huddersfield, Huddersfield HD1 3DH, UK
impress@hud.ac.uk

Abstract

In this paper we describe a tools environment which automates the validation and maintenance of a requirements model written in many-sorted first order logic. We focus on: a translator, that produces an executable form of the model; blame assignment functions, which input batches of mis-classified tests (i.e. training examples) and output likely faulty parts of the model; and a theory reviser, which inputs the faulty parts and examples and outputs suggested revisions to the model. In particular, we concentrate on the problems encountered when applying these tools to a real application: a requirements model containing air traffic control separation standards, operating methods and airspace information.

1. Introduction

A unifying theme in the research areas of knowledge engineering, requirements engineering and formal methods is the construction and validation of requirements models represented as formal systems (using languages such as RML [7]). Within the knowledge based system community, formal specification has been hailed as providing a bridge between the conceptual models of informal knowledge acquisition methods (such as KADS [1]) and implementations of knowledge-based systems [22], as well as being important in the verification and validation of KBS [15]. Even in areas such as AI Planning, the construction and validation of a domain model is recognised as a critical step towards the construction of a final system [13]. Within Software Engineering it has been argued that the use of formal specification and formal methods can produce many advantages for system development. Establishing a detailed set of requirements in such a precise form supports automated analysis of those requirements via logical deduction, or in some cases prototyping. The potential for automation of the software development process starting with a formal model is emphasised by for example Shaw and Gaines [19]. Here they describe the advantages in 3 parts, as a move towards proof of correctness of implementations, simulation of requirements to support specification development, and automatic generation of efficient implementations. This has been backed up by a number of large-scale applications, especially in the safety-critical areas (for example see reference [10]).

Given that the production of a precise, abstract domain model is desirable, a prime concern is the validation and maintenance of the model i.e. ensuring that it is kept accurate and complete. Validation of a *formal* model has problems and advantages: it may be harder for a non-computing professional to understand, and be more detailed than a conventional requirements document [17]. On the other hand, the formality brings with it the opportunity for powerful tool support, in particular, animation [16]. In any case, such a model 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 (or of the *mediating specification*, in the terminology used in reference [19]).

The work reported here has been carried out and driven by a particular application - the initial capture, validation and maintenance of the knowledge intensive requirements of an air traffic control system. The requirements represent the separation criteria and conflict prediction method for air traffic management in the North East Atlantic. The corresponding model is written in many-sorted logic[1], and has been encased in a tools environment which, to some extent, automates the validation and maintenance process. To achieve this, we have had to overcome the problems of fielding tools originating from the area of Artificial Intelligence.

The paper is organised as follows. In section 2, we will give a brief introduction to the application, and describe the

[1] it is recorded in the 'Formal Methods Europe Applications Database' web site http://www.cs.tcd.ie/FME, and its initial capture and tool support are detailed in reference [14]

formal model that was created to represent it. In section 3 we describe the central tool in the environment, an animator which translates the model into an executable form, and allows one to 'test' the requirements. In section 4, we describe the blame assignment and theory reviser functions, which input batches of training examples, identify parts of the model that are most likely to be faulty and output suggestions for revisions to the model. In section 5, we briefly discuss other processes that have been used to remove bugs from the model.

2. The ATC Domain Model

2.1. General 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'. A specification of a profile consists of a sequence of (roughly) five or six 'straight line' segments, as well as the call sign, type and certain characteristics of the aircraft. 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.

This requirements model is intended 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 [6], where a tabular style of specification and a variant of higher order logic is used.

2.2. Producing a Customised Model

An important criteria in developing a requirements model is to keep the 'semantic gap' between application and model as small as possible. This allows the model's notation, or an equivalent 'pseudo-natural language' form, to be understandable to non-computing professionals (overcoming at least to some extent Leveson et al's criticism of the use of formal systems in requirements analysis [10]). We chose Many Sorted First Order logic (here abbreviated to 'msl') to encode the model for a number of reasons, detailed in [14]. As msl is a very general language we customised it chiefly through the imaginative and precise use of syntactic constructs. In our ATC model, all the terminology was chosen to fit in with the source terminology, and our resulting model is readable and understandable by air traffic controllers[2].

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

(1) The **object level** consists of a set of msl axioms, directly describing the objects, object classes, 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 model is hierarchical in nature. As shown in the figure, axioms are used to define the domain object classes (i.e. the *sorts* in msl) serving as primitives in the model, as well as the higher levels containing relations between and functions of the domain objects, and factual information specific to the application. A typical instance of the requirements model contains over 2000 axioms - over 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).

Levels 1,2,3,5 and the non-transient airspace information is collectively referred to as the the 'CPS' (conflict prediction specification) and is encased in documented files shown in Figure 2. An axiom from the Auxiliary set is given in Example 1 together with its English paraphrase. It defines a geometrical relation between parts of an aircraft's profile, and we will use it throughout the rest of the paper.

[2] though they found our lower level object and geometrical axioms - not surprisingly - quite tedious to read

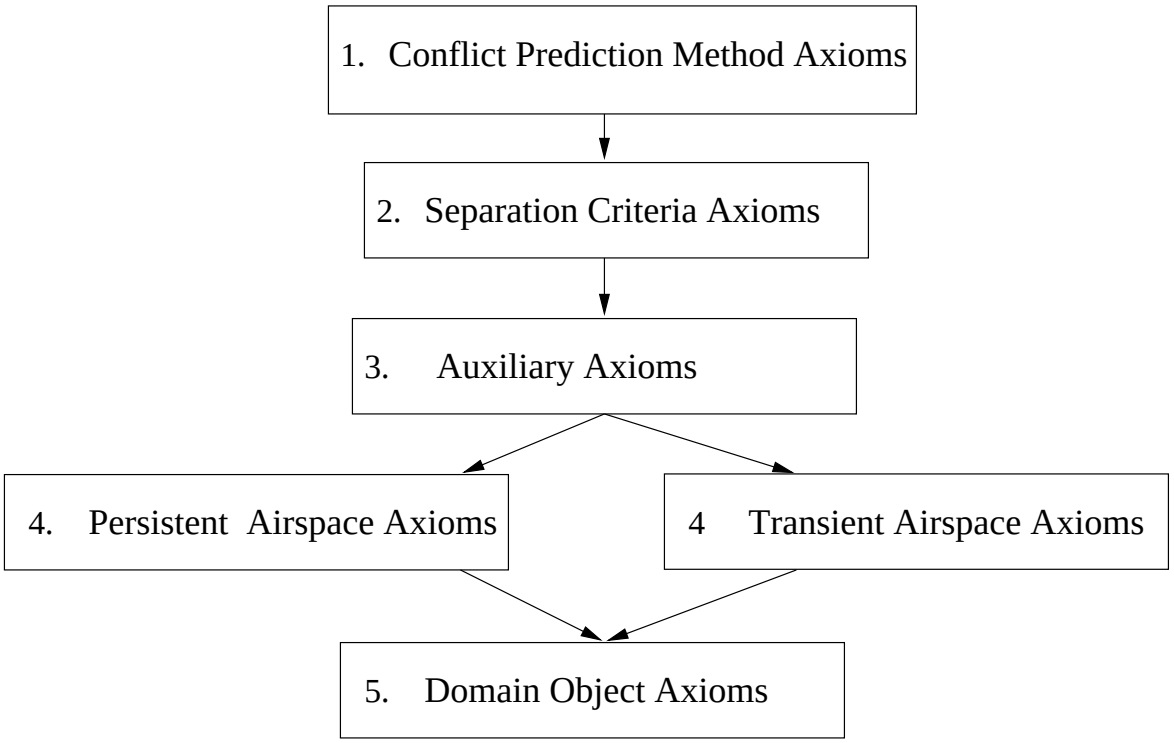

Figure 1. Axiom Structure in the Requirements Model.

```
"(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)
```

Example 2 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-model** is composed of (a) the precise syntactic specification of all the phrases and atoms in the model (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-model are explained in later sections. Part (a) is the main component of the meta-model 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. This grammar also forms one of the tools in the environment (it is part of tools *msl*2*EF* and *msl*2*VF*) as the Prolog interpreter animates it to form a parser for the model. The grammar is *two level*: the top level is model-independent, defining the logical connectives, whereas the lower level contains the concrete syntax which customises the model. Example 3 is a rule that defines the syntax of the mixfix predicate used in Example 1, where Segment1 and Segment2 are correctly formed terms of the object class

```
"(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

'segment', defined in another part of the grammar. Text appearing literally in the model appears in square brackets. Each legal form of each object class 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.

3. Animating and Testing the Requirements Model

Validating and maintaining a formal requirements model is a complex and a repetitive task, and so automated tools to assist the process are considered essential. The method we advocate is iterative - the inputs to the validation tools are source documents containing the model (shown in the upper dashed box of Figure 2). Additionally, 'training data' in the form of test data, and specific queries and properties, expressed as theorems, are required.

The outputs of the processes described below are analysis reports, and suggestions for revisions. Revisions are carried out if none of the validation processes indicates a bug in, or a bug resulting from the revision; revisions are carried out by changing one or more of the CPS's formulas, the components of the meta-model, and their associated documentation.

In the following sections we will describe one iteration in the process, and the tools used to assist in this. In doing so we will concentrate on the general tools and only briefly describe tools that are specific to the ATC application (e.g. data preparation and graphical simulation).

3.1. The Animator

Raw test data is translated into msl using the *data2msl* process, forming the transient aircraft and airspace information (Example 2 is an example of an axiom 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 the Atlantic on a certain day. Likewise, the documented theory is processed into a stream of formulas by *tex2msl*.

The tool labelled *msl2EF* in Figure 2 is vitally important as it produces a faithful *operational* form of the model (*EF* in *msl2EF* means 'execution-form') which is used in the testing and theory revision processes discussed below. The model enters *msl2EF* as a stream of formulas, and each formula is checked sequentially against the stringent syntactic definition residing in the meta-model. If the syntactic checking process passes without error, then the translator continues by translating the formula into a clausal form, respecting the syntactic conventions of Prolog. The translator ensures that each clause form logically follows from the original axiom in the CPS. Formulas written in the customised msl are not restricted logically, in the sense that variables from object classes can be universally and existentially quantified over sorts, 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. There are some restrictions, however, 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.

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 Example 5 below.

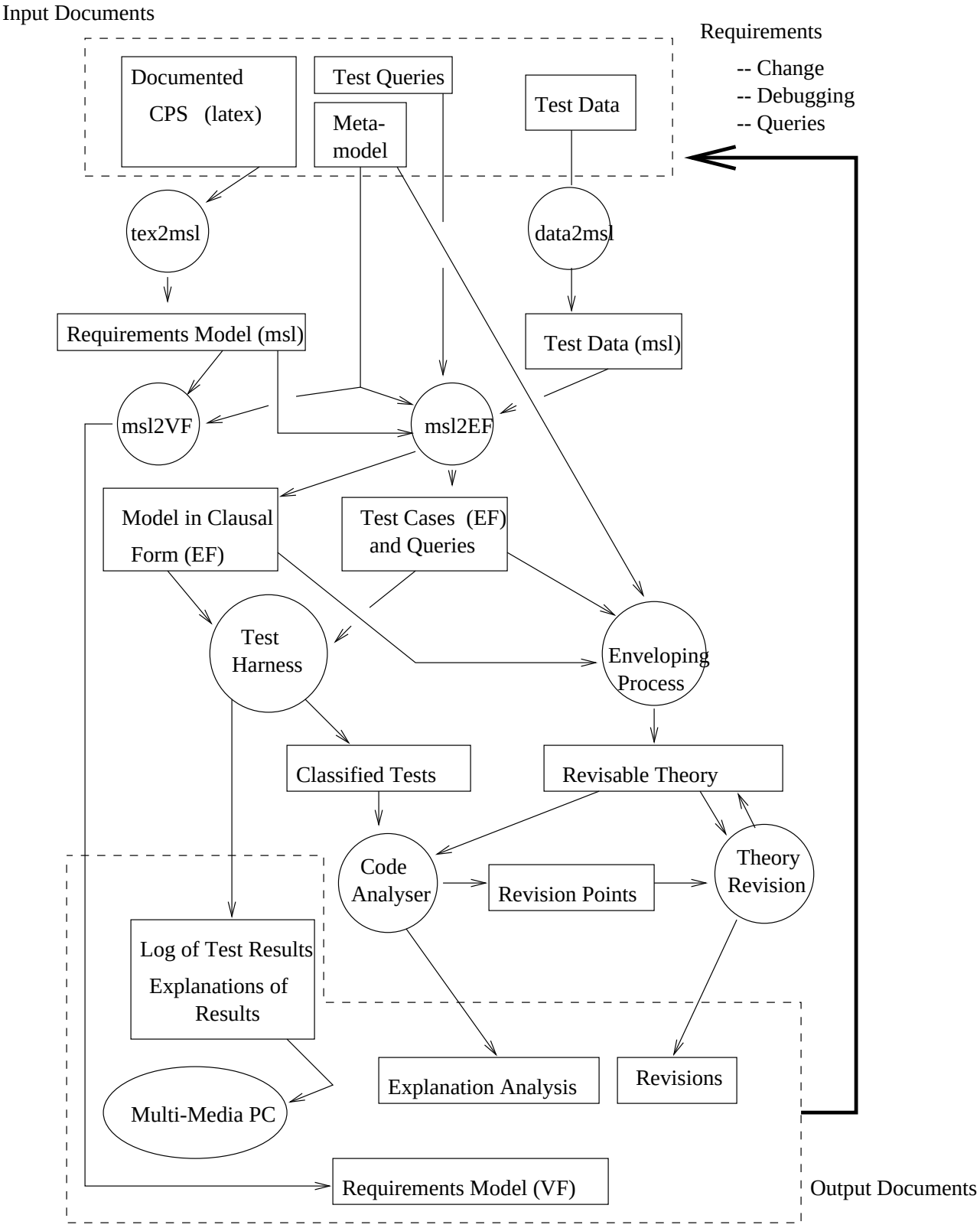

Figure 2. The CPS Tools Environment.

```
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, !.

Example 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.

3.2. Testing the Model using its EF

Executing an ATC Simulation

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 2) as a series of data sets, each set representing the characteristics of a 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 air traffic control officers. The Test Harness executes the definition of main relation repeatedly, systematically checking pairs of profiles (below this predicate is referred to as the 'conflict relation'):

'profiles_are_in_oceanic_conflict(P1,P,S1,S)'

Each profile P1 is compared with all[3] other profiles, P. For given P1 and P, if this relation is

– *true*, there are at least two segments (S1 from P1, and S from P) that are in conflict, in which case P1 and P are deemed to be in conflict

– *false*, then the P1 and P are separated to the required standard, according to the CPS.

Mismatches between the expected result and the result returned by the CPS in EF form drive the revision processes explained below.

Executing Specific Queries

A set of queries (which are 'distinguished tests') can be built up over the lifetime of the model. 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 follows:

Pair:	CPS Result:	Classification:
(Query, True)	True	Truely Positive
(Query, True)	False	Falsely Negative
(Query, False)	True	Falsely Positive
(Query, False)	False	Truely Negative

Queries may consist of calls to the conflict relation, or of any other lower level predicates or functions in the model. 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 process explained below. ATC simulation 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,P,S1,S), False)

where S1, S are existentially quantified over the Segment's sort, and P1 and P are profile identifiers.

4. Automated Debugging and Maintenance of the Requirements Model

4.1. The Revisable Theory

If a test run has produced misclassifications, the Enveloping Process can be executed. This inputs the clauses making up the CPS and the transient test data in EF, and outputs what we call the Revisable Theory denoted CPS_{RT}. This in an 'enveloped' form of EF, where every clause $\mathcal{C}$ in the theory has the form:

fact(*C*, index, validity level)

where index identifies the clause, and validity level is either

– 'shielded', meaning the clause is to be left unchanged, or

– 'revisable', meaning that the clause is a candidate for change.

– 'unrevisable', meaning that the clause and all its goal definitions are to be left unchanged.

The validity level of a clause is the same as the axiom from which it originated in the CPS, and is held in part of the meta-model.

The Code Analyser executes CPS_{RT} using techniques based on meta-interpretation [21]. For individual test

[3] 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

Form a set of pairs P of the form '$(i, 0)$',
where i ranges over the indices of the clauses in CPS_{RT};
for each falsely positive instance:
    generate a proof tree;
    for each clause instance C in the proof tree:
        obtain C's index j;
        overwrite (j, n) in P with $(j, n + 1)$
    end for
end for

Figure 3. Outline of Blame Assignment Algorithm

runs where a goal predicate succeeds, this process produces an Explanation Analysis file (see Figure 2) using an explanation-based generalisation technique derived from that published in [9]. Whereas revision techniques in the KBS community have been used in a kind of incremental fashion (e.g. as in the MOBAL system [20] or the KRUST system [4]), 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.

4.2. Blame Assignment

The main part of the Code Analysis process is *blame assignment*, i.e. the use of successful proof trees (and failed proof traces) to indicate the parts of the CPS_{RT} that are likely to be faulty. The use of revisable/shielded/unrevisable distinction above means that proof trees are truncated and proof trees of predicates defined by shielded or unrevisable clauses are not recorded. Factual axioms, axioms that are considered sound through exhaustive visual inspection, and numeric functions implemented via the Prolog interpreter, are all labelled in the CPS as shielded. Hence Example 1 is revisable, whereas Example 2 is shielded.

A simple blame assignment algorithm for falsely positive instances is shown in Figure 3. This algorithm terminates indicating the most likely clause for revision as the one with index j, where $(j, n) \in P$ and n is the maximum value recorded. Standard meta-interpreters for generating proof trees, however, are restricted to definite programs [3, 21], and cannot cope with programs which include negative literals. We required a technique which could generate proof trees which explicitly represent negation in general logic programs. To cope with the expressiveness demanded by the real application, we extended the meta-interpreter technique for proof tree generation to negative literals. This extension was based on Clark's notion of the completion of a general logic program, which he introduced to justify the use of negation as failure rule [5]. In our work, the explicit representation of negative clauses is achieved by first unfolding negative literals and then transforming them using De Morgan's laws (the technical details can be found in [23]). In consequence, a blame algorithm which takes into account 'negative trees' is more accurate than one in which negation is automatically shielded. In contrast, Wogulis [24] has developed a system that can revise first-order theories containing errors within the scope of a negation; however there are not sufficient details in the paper to determine whether it will scale up cover our application.

```
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)) ),!.

Example 6
```

As an example of blame assignment, we describe an experiment carried out to update the CPS. The separation criteria for aircraft profiles in the Atlantic is updated from time to time, for example to take into account new navigational aids and equipment. Recently a certain class of aircraft have been allowed to fly with a 'reduced separation' between certain flight levels i.e. the vertical separation criteria have been relaxed, meaning that the conflict relation has been specialised (in a machine learning sense). We obtained a day's worth of flight profiles, cleared using this new criteria. After a batch run of 5040 profile comparisons using the conflict relation, we obtained 96 falsely positives resulting from the mismatch in vertical separation criteria. The blame algorithm was run with the negative examples and pinpointed several clauses where that were likely to be faulty. One was the clause defining the vertical separation criteria for 2000 feet, shown in Example 6. This clause was then input to the theory revision process discussed below.

4.3. Theory Revision

Animation shows the presence of bugs, blame assignment indicates the likely part of the theory in which they reside, and theory revision (TR) minimally updates the theory to eradicate the bugs. These revisions are then used by an engineer to help update (with results from other validation tools) the requirements model.

A TR algorithm systematically uses TR *operators* to alter a clause by adjusting, removing or replacing the liter-

als making up that clause [25, 18]. Three simple TR operators are 'delete clause', 'delete antecedent' and 'add antecedent'. An exhaustive search strategy which explores the effect of revision operators would obviously falter due to the immense combinatorics of the process, and even with fairly restrictive assumptions recent theoretical research has shown that theory revision is hard [8, 2].

Our experiments with algorithms involving the use of simple TR operators confirmed the complexity problems, but after introducing some general assumptions we achieved a certain amount of experimental success. Firstly, the shielding process allows part of the theory to be held fixed. Out the 2000 clauses or so making up CPS_{RT}, we typically kept the 'top' 122 clauses revisable, (these defined the most complex relations where bugs were most likely to be) and the rest were shielded. Secondly, we implemented a form of *focussed* revision, whereby the revisions could only be carried out to *ordinal* literals only, i.e. those literals which specify a relation between objects of an ordered sort. Details of the focused revision can be found in reference [12]); we present a summary here.

Associated with each ordinal sort is a binary, transitive, ordering relationship (predicate) we call '$\succeq$'. The CPS contains over 200 instances of such geometrical, ordered relations, and these often make up the most complex conditions. Example 6 contains two such literals,

'one_or_both_are_above'
'one_or_both_are_at_or_below'.

Each clause $\mathcal{C}$ in the CPS_{EF} has its *variable* domain:

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

where each X_i is an ordinal sort and each D_j is a nominal sort (viz. *not* ordered). If we factor out the X_i from the D_j components, then for each clause, and for each tuple $\mathbf{d}$ of values $(d_1, \ldots, d_m)$, there is defined an n dimensional region $\mathcal{R}(\mathbf{d})$ a domain of applicability of the clause. Since the clauses may involve disjunction, then there may be several different values of $\mathbf{d}$ associated with a clause. Thus region $\mathcal{R}(\mathbf{d})$ associated with a clause $\mathcal{C}$ is populated by n-tuples $\mathbf{x}$ of Prolog variables, where each component variable of $\mathbf{x}$ is ordinal. Revisions of $\mathcal{C}$ consisted of revisions of its region $\mathcal{R}(\mathbf{d})$ to $\mathcal{R}'(\mathbf{d})$ so that it was populated only by correctly classified instances. Revisions operators were of two kinds:

Simple operators involve deletion and addition of antecedents from a clause, as in conventional TR, although the antecedents are restricted to occurrences of order relations. We implemented a TR algorithm with simple operators based on these assumptions, and found it capable of spotting and removing simple bugs involving revisions in one or more clauses. This is akin (in 2-D geometrical terms) to examining reflections of the region about a straight line.

Composite Operators involved identifying a sub-region $\Delta\mathcal{R}(\mathbf{d})$ associated with maximum and minimum values of incorrectly classified instances. $\Delta\mathcal{R}(\mathbf{d})$ was either *deleted* from or *added* to $\mathcal{R}(\mathbf{d})$ according to whether the instances were falsely positive or falsely negative.

```
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),
          (( not__(A is_at_or_above fl(330))
            ; not__(A is_at_or_below fl(370)))
            ; not__(C is_at_or_above fl(330))
            ; not__(C is_at_or_below fl(370)))
        ; C is_above fl(290),
           (( not__(A is_at_or_above fl(330))
            ;  not__(A is_at_or_below fl(370))
             )
            ; not__(C is_at_or_above fl(330))
            ; not__(C is_at_or_below fl(370))))
         ;
 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 7
```

Composite Operators involved identifying a sub-region $\Delta\mathcal{R}(\mathbf{d})$ associated with maximum and minimum values of incorrectly classified instances. $\Delta\mathcal{R}(\mathbf{d})$ was either *deleted* from or *added* to $\mathcal{R}(\mathbf{d})$ according to whether the instances were falsely positive or falsely negative.

Example 6 was revised with a 'composite' operator, using the results of the blame assignment discussed above. After approximately 2 days of processing, the TR algorithm found an updated clause $\mathcal{C}$ which gave a 100 per cent accuracy classification, and is shown in Example7.

This was an exciting result as it appears to encode the new criterion for 2000, effectively excluding the region between 33,000 feet and 37,000. (These were the minimum and maximum values of flight level variables associated with falsely positive instances of $\mathcal{C}$ for subsonic aircraft.) However, it loses the readability quality of the hand-crafted model.

5. Conventional Debugging and Maintenance of the Requirements Model

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 bugs which appear in the environment of the CPS, rather than in the CPS itself. The procedures are as follows:

(a) model inspection: The model can be translated to validation form, using tool *msl2VF* shown in Figure 2. The

mapping given in the meta-model, 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 [14], and experience has shown that this is most successful in the initial validation of the model, and for removing bugs in the top level axioms.

(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 a bug 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 [11]. This particular tool helped us spot 'noisy data'. Inspection of several apparent 'falsely positives' using the display clearly showed that the flight profiles were in fact in conflict. The bug 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, 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 bug which resulted from the animation process, and occurred in an auxiliary axiom relating the speeds of two aircraft (shown in Example 8).

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.

```
" [ (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 8
```

6. Conclusions

We have described an environment which has been used for the debugging and maintenance of a customised requirements model in many-sorted logic. We have shown the feasibility of using advanced techniques such as theory revision on specifications of the order of the CPS, although we had to cope with problems of expressiveness in that we had to extend tools to cope with logic clauses containing negation in their bodies, and arbitrary functor structures within literals. Also, we had to overcome the problem of scale in theory revision using a focused version which is restricted to revising literals representing ordering relations. In our current project we have used (b), (c) and (d) in section 5, together with blame assignment and theory revision, to cut the error rate by a factor of 20 - from approximately 200 to 10 errors in every 10,000 tests.

Problems with our current application and tools include:

– we have concentrated on techniques inputing a skewed set of training data. While the number of negative examples of the conflict relation are virtually limitless (using historic records), we could only obtain small amounts of positive examples.

– although there is a direct mapping between axioms in the CPS and clauses in CPS_{EF}, revisions suggested by TR have to be re-engineered into msl form for insertion into the CPS. As can be seen by Example 7, the revisions are not easily understandable.

Our future work will concentrate on these problems and the further evaluation of our environment, in particular the development of a generic version for use with other requirements models stated in their own customised form of msl. We also intend to further explore the limitations and possibilities of theory revision. For example, in the TR process one could keep the whole CPS_{TR} shielded and input flight profiles that were in conflict represented by revisable axioms. Using theory revision applied to the clauses representing these flight profiles the reviser might return new clauses representing a cleared profile, i.e. perform a process of conflict resolution.

Acknowledgements

This project is supported by an EPSRC grant, number GR/K73152. We would like to acknowledge the help of Chris Bryant, who implemented some of the theory revision 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

[1] 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.

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

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

[14] T. L. McCluskey, J. M. Porteous, Y. Naik, C. 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.

[15] P. Meseguer and A. 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.

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

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

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

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

[20] 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.

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

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

[23] 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.

[24] J. 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.

[25] 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.