ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

D Duke, University of York, UK and A S Evans, University of Bradford, UK (Eds)

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop, Ilkley, UK. 23-24 September 1996

The Validation of Formal Specifications of Requirements

T.L. McCluskey, J.M. Porteous, M.M. West and C.H. Bryant

Published in collaboration with the British Computer Society

BCS FACS

Springer

The Validation of Formal Specifications of Requirements

T.L.McCluskey, J.M.Porteous, M.M.West and C.H.Bryant
The School of Computing and Mathematics, The University of Huddersfield,
Huddersfield, W.Yorks, UK

Abstract

We review the approaches put forward to validate formal specifications of requirements, drawing a parallel with research into the validation of knowledge bases. Using an industrial-scale case study we describe a partially implemented, integrated environment for validating requirements stated in many-sorted first order logic. In particular, we show how techniques from machine learning can be used to provide extra tool-support for the validation process.

1 Introduction

The issue of validation of formal specifications seems to have received rather less attention than other processes in specification development such as verification and reification [16]. It may be because these latter processes are more amenable to the use of mathematical calculi, and therefore more attractive to explore; or it may be that validation is usually associated with other areas of software development not covered by formal specification. Whatever the reason, promotion of the accuracy of a formal requirements specification as a *model* of the informal requirements is arguably one of the most important activities in the development process.

The main approach taken in the research area of automated validation of formal specifications has been to test an animated version of a specification with batches of test cases in a similar manner to program testing. Given that the specification can be faithfully and efficiently prototyped, a problem with this approach is that *failure* of tests in a batch does not by itself pinpoint or repair the faulty or missing part(s) within the original specification that led to the failure. A goal of our current work is to integrate a 'conventional' tool - supported validation environment with techniques from machine learning (ML), a branch of artificial intelligence that covers such areas as knowledge acquisition and knowledge refinement. In ML terms, a formal specification of requirements can be viewed as a developing domain theory. A major research area in ML concerns processes using deductive and inductive techniques to repair domain theories [2, 17]. In our work we are exploring the use of ML techniques to create novel methods for the automated validation of formal specifications.

In section 2 we review some of the current work in validation of formalised requirements. In section 3 we describe a case study concerning the development of a formal requirements specification carried out on behalf of the National Air Traffic Services Ltd as part of the FAROAS project [13]. We focus on the validation stage, detailing the five processes that were used to promote its accuracy. In section 4 we discuss the connection between the validation of knowledge bases and the validation of formal requirements specifications, and conclude that there is a parallel between the research in both areas. In section 5 the reader is introduced to the area of machine learning. Using the case study as an exemplar, and starting from the premise that a requirements specification forms an evolving theory, we show how test cases can be interpreted as positive and negative instances to which the theory should conform. Hence we describe how automated theory revision techniques can be used to revise the specification, and complement the conventional processes in the validation environment.

2 Approaches to the Validation of Formal Specifications

It is, in principle, possible to prove that a delivered software system will execute in a manner which satisfies the formal specification, and this process is one of many possible *verification* activities in system development. In contrast, it is not possible to prove that a formal specification of a system correctly captures its informal requirements: the validation process concerns the accumulation of evidence that this is so [16]. *Validation* [20] is derived from the Latin 'validate' meaning 'to confirm or ratify', and requires an informed decision to be made.

Figure 1 shows a model of system development that emphasises our concern with the validation of a formal specification of a system's functional requirements (as opposed to validation of an implemented system). Here *domain knowledge* is acquired from several or all of the various information sources. These may include (i) stakeholders [9] such as system experts, users and managers (ii) manuals, standards and training literature of an existing manual system (iii) specification and design documentation of an existing software system (iv) typical or benchmark test cases of system operation. Although the form of validation will necessarily have to differ depending on the importance of each of the sources, the dominant form of validation in software engineering has been user inspection of the generated requirements documents. This may provide necessary evidence of successful capture with conventional text, but requirements formalisation would seem to make user inspection more difficult (in the sense of introducing a mathematical language). On the other hand an advantage of formalisation is that it allows the possibility of a host of automated techniques to improve validity. Chief among these is the possibility of animation of the specification and then testing this executable prototype with a suite of customer-supplied tests.

Research into the use of tools to help the validation process includes the work of Mukherjee [16]. Here he reports the use of a CASE tool for syntax and type analysis, and specification execution, which helps the validation of VDM specifications, and he shows that several types of error can be uncovered using these automated forms of validation. A development model for safety-critical software which includes similar activities is provided in an IEE/BCS report [23]. In this development process the results of the animation are set aside and compared with the final tests for the equipment. In order for satisfactory validation, the specification needs to be able to stand up to critical examination by the customer (or user). Rushby [19] calls this *challenging the specification*. "The idea of a challenge is to pose a question that an adequate specification should be able to answer." It might be thought that the derivation of theorems from a formal specification would satisfy the requirement of such a challenge. However a problem with such methods is the need for the 'expert' in theorem proving to reformulate and often generalise the user's informal queries [22]. The equivalence between the expressed query and the formal query cannot be proved; the logic of the formal specification is often visible only to the expert. Constructing and proving conjectures about specification properties is both time consuming and expensive and if the goal is to find the right theorem and prove it quickly, it might be best to search for counter-examples rather than a proof [8].

Although formal reasoning and animation are therefore powerful techniques for use in the validation process, used in isolation they are likely to detect only certain types of errors. In the next section we show how a *diverse* set of strategies has proved useful in uncovering different types of bugs during the validation of functional, technically-oriented requirements. Given the wide ranging types of bugs in requirements specifications, from syntax or type errors, to complete omissions, a systematic combination of various (automated) validation techniques appears to be the best strategy.

3 A Case Study

3.1 The CPS: its structure and content

'Shanwick' is a large area of airspace in the eastern half of the North Atlantic, managed by air traffic control centres in Shannon, Ireland and Prestwick, Scotland. In a previous project called FAROAS a prototype formal requirements specification for the *conflict prediction* of aircraft flight profiles through the Shanwick airspace was developed, in a manner as indicated in Figure 1. Domain knowledge was elicited from each of sources (i) - (iv), the majority coming from (ii). The resulting specification, called the *CPS*, is structured into six groups of axioms representing domain objects, persistent airspace information, daily flight information, aircraft separation criteria, conflict prediction procedures, and a set of auxiliary axioms defining general technical terms and conditions[1]. A formal requirements engineering environment (referred to as the FREE) was created to help develop the CPS, and at the end of the project

[1]The domain model encasing the CPS contains other axioms such as those modelling the resolution of aircraft conflicts.

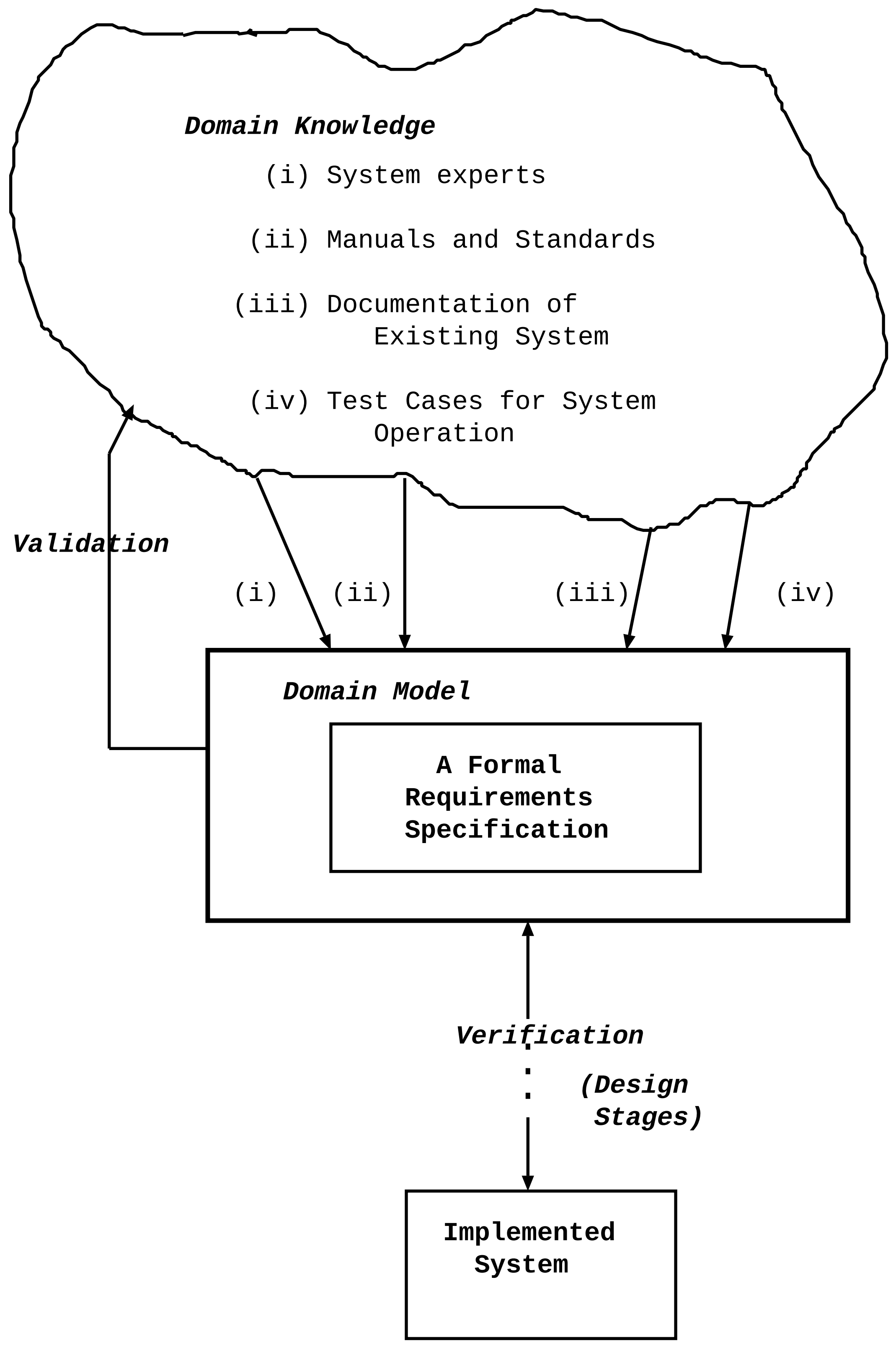
Domain Knowledge
(i) System experts
(ii) Manuals and Standards
(iii) Documentation of Existing System
(iv) Test Cases for System Operation
Validation
(i)
(ii)
(iii)
(iv)
Domain Model
A Formal Requirements Specification
Verification
(Design Stages)
Implemented System

Figure 1: A Model for Requirements Capture.

a version of the CPS containing more than 1000 axioms had been created. More details of the work can be found in reference [13].

All the axioms in the CPS are written in many-sorted first order logic (MSFOL) enriched with real and natural numbers. The *domain object* axioms are used to give meaning to sorts, and are encapsulated with the definitions of a sort's constructor and selector functions in syntactic units. Sorts include Profiles (representing projected flight paths), Segments (components of Profiles), Time, Aircrafts, 2-D points, 3-D points, 4-D points and Flight Levels. The Segment is the most complex sort, having 44 primitive function definitions and 16 primitive predicate definitions, together defined by 60 axioms. A value of type Segment:

```
the_Segment(profile_ELY350_1,
            52 N ; 40 W ; FL 350 ; FL 350 ; 02 05 GMT day 0,
            53 N ; 30 W ; FL 350 ; FL 350 ; 02 51 GMT day 0, 0.83)
```

is an aggregation of four values of other sorts which are a profile identifier, two 4-dimensional points representing the start and end of a segment (4D points require 5 components as each has an input and output flight level) and a speed expressed as a mach number. This example denotes a segment where an aircraft is planned to fly in at 35000 ft, stay in level flight at mach 0.83 for 46 minutes, then exit the segment at the same flight level. Note that 'the_segment()',';', 'FL','N' etc are all interpreted as algebraic constructors, and the inclusion of a profile identifier allows the possibility of two distinct segments from different profiles to have identical attributes.

The CPS contains several hundred *airspace information* axioms particular to the Shanwick sector, containing aircraft and airfield details; three typical examples are:

```
"(FL 55 is_the_min_flight_level_for Shanwick airspace)"
"(B747 is_a_jet_type)"
"(63 59 N ; 22 36 W is_a_prox_airfield_pt)"
```

The *flight information* axioms are less permanent, describing the details of daily projected flight paths in terms of Profile identifiers and values of Segment.

The CPS contains several hundred 'high level' axioms which mirror complex definitions and procedures, mainly taken from the Manual of Air Traffic Services, Part 2[1]. An example of an *aircraft separation* axiom, which defines a vertical separation value for aircraft flying in the upper airspace, is as follows:

```
AXIOM 3.1.2
"  [(Segment1 and Segment2 are_subject_to_oceanic_cpr) &
    (Flight_level1 lies_in_flight_level_range_of Segment1) &
    (Flight_level2 lies_in_flight_level_range_of Segment2) ]
   =>
   [(the_min_vertical_sep_Val_in_feet_required_for
     Flight_level1 of Segment1 and Flight_level2 of Segment2) = 4000
    <=>
   [(one_or_both_of Segment1 and Segment2 are_flown_at_supersonic_speed)
     & (both Flight_level1 and Flight_level2 are_above FL 430 )]
   ]".
```

The customised language used for the domain was chosen to reflect the knowledge sources. This was to prove particularly helpful for Air Traffic Experts (ATCOs) during hand validation sessions. Hence the use of mix-fix predicates and functions such as:

```
     one_or_both_of ...  and ... are_flown_at_supersonic_speed
```

This predicate is defined in turn by an axiom in the set of *auxiliary* axioms.

The domain model language was constructed using a large set of grammar rules which give a customised form of MSFOL. The predicates and their strongly typed arguments (either constant identifiers, variable identifiers or function identifiers with strongly typed arguments) that can appear in any sentence of the CPS were thus precisely defined. We call this grammar, which was implemented using the Prolog grammar rule technique, the $G(CPS)$.

3.2 Validation of the CPS

Although the ATC domain model is large, highly structured, and complex, there were certain assumptions that made the validation of the CPS tractable. Firstly, the main function required, that of conflict prediction, is essentially a predicate. Aircraft profiles are either separated or they are not. Secondly, the model of conflict prediction does not need to take into account state changes, actions or agents as the airspace is modelled as a static four dimensional snapshot. Finally, after the first version of the CPS had been built, it was discovered that the axioms were structured in an hierarchical fashion, allowing the conflict prediction predicate to be operationalised via a (fully automated) translation of the CPS into an executable clausal form.

On the other hand, the fact that some knowledge had to be elicited from domain experts raised special problems for the validation of the CPS. In particular there is rarely full agreement among experts, and their understanding of the domain tends to change over time. Hence the optimal solution seemed to be to promote the ‘fit’ between model and domain in various ways, whilst allowing efficient means of model maintenance. Two important features of the validation processes included their diversity and the fact that they were well supported by tools. The tools environment that was built to surround the CPS is shown in data flow terms in Figure 2 and is adapted from figure 5 in [13]. Using Figure 2 we describe each of the five processes that could uncover errors in the CPS, where boxes represent a document or datastore, and ovals represent processes.

Process 1 checks every part of the CPS against G(CPS) and either outputs grammatical errors, or if it is successful, builds a parse tree of each sentence (axiom). A translator then uses this parse tree to create a version of the CPS in execution and ‘hand’ validation forms, which are in turn input to the processes below.

Process 2 inputs the executable form as a general Prolog program. If the CPS contained a set of queries (existentially quantified formulae) then these are translated to Prolog queries and interpreted as tests. The test harness executes the queries, and compares the results output with expected results. The most important queries involve the execution of the conflict prediction predicate itself. In this case the flight information axioms are generated from expert-derived tests (typically involving two flight profiles in close proximity) and encoded as part of the CPS.

Process 3 refers to manual validation, involving the study of the ‘hand’ validation form of the CPS by domain experts. This form consists of structured English sentences and diagrams showing the relational structure of the axiom set.

Process 4 takes the form of reasoning about the specification’s internal consistency. A proof of the overall consistency of the CPS is performed by constructing a particular interpretation for it which satisfies each of its axioms. Also, as part of this process, we have also carried out proofs that certain properties of the CPS hold.

Process 5 An interface was produced that allowed air traffic control experts to input flight profiles, run the conflict prediction function and request explanations of conflict decisions. In the event of a detected conflict between two flight profiles the simulator can, if required, identify the segments which were in conflict, and indicate the required separation values that were violated.

Each of these five validation processes uncovered errors in the initial encoding of the CPS, and these were fed back to the modellers who made the required changes. The use of such a diverse range of strategies is particularly effective in improving the accuracy of the model when used in combination. For example, changes could be made to the CPS after a session involving Processes 3 and 5, and then these could be immediately cross checked using Process 1 and 2.

There still remain, however, particular problems with these strategies. Identifying bugs and repairing the specification as a result is relatively straightforward using the manual Process 3. Here the *declarative*, compositional form of the requirements is exploited, giving a fairly straightforward semantics to each part of the CPS. Unfortunately, manual inspection is also painstaking and very labour-intensive, and validating the whole of the CPS this way proved impracticable. In contrast, the validation processes involving execution of a derived prototype are fully automated, and therefore can be easily re-executed after maintenance of the CPS. On the other hand, while these processes can show the presence of bugs, they may fail to pinpoint them, and they certainly do not help to repair them. This is because execution involves the combination of many parts of the CPS and hence it is difficult to assign blame when a bug is detected (i.e. in the case of an expected result not matching with an actual result). For domain models containing specifications at least as complex as the CPS, there is certainly a need for more powerful, automated validation tools to help overcome these problems.

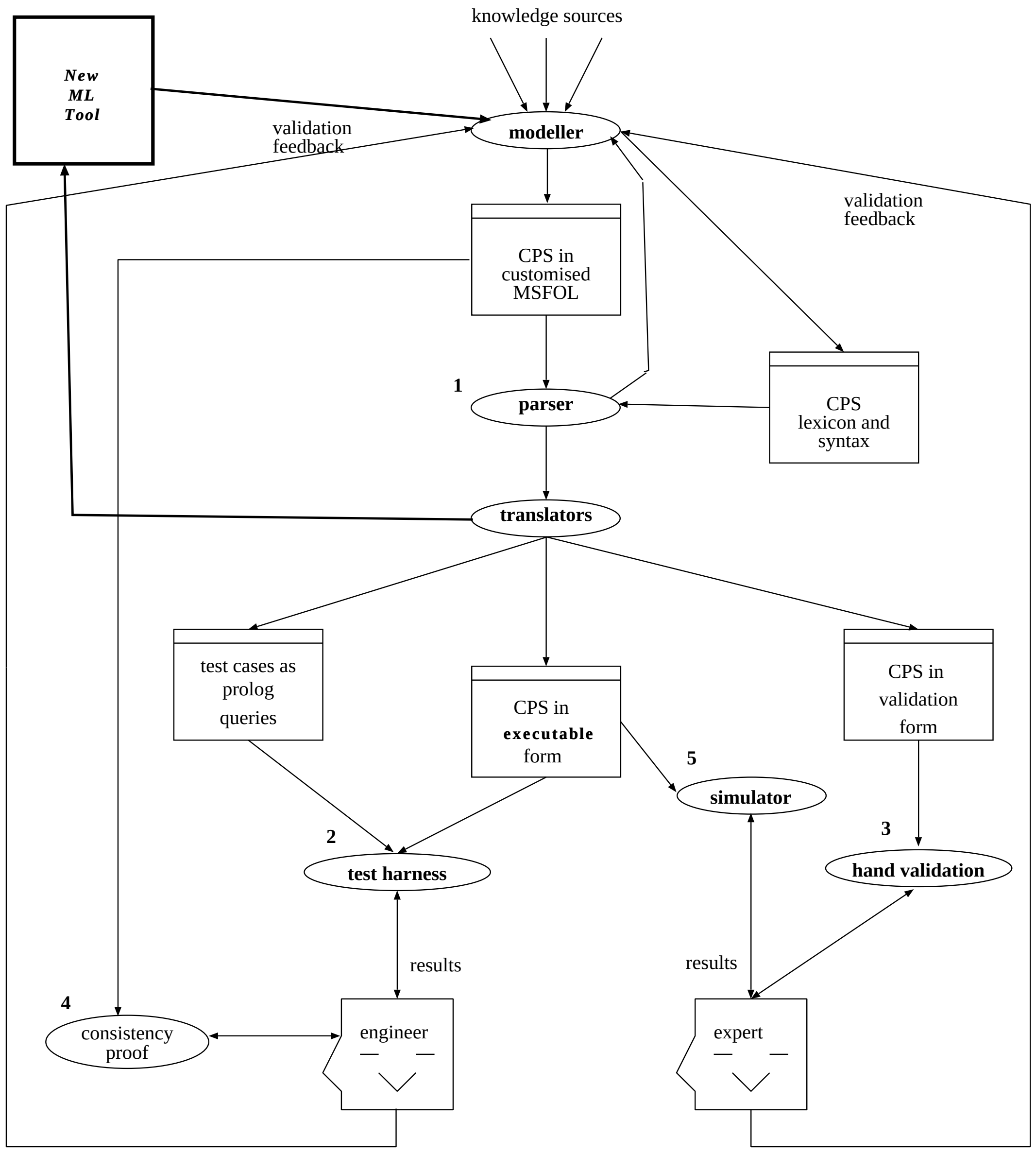

Figure 2: The FREE

4 Viewing The Domain Model as a Knowledge Base

Figure 1 emphasises the connection between knowledge acquisition and requirements capture by showing a ‘domain model’, which is a body of elicited knowledge from which the requirements for a particular system is crystalised. With this in mind we can explore the connection between functional requirements stated in some formal logic, and the knowledge base of a knowledge-based system (KBS). In technically-oriented areas in particular, we observe that a logic-based formal requirements specification (FRS) can be interpreted as “a large body of specialised knowledge in a narrow domain” and so can be viewed as a “knowledge base” [24]. The major differences would seem to be:

- differing roles: a KBS’s knowledge base is just one of several components of a system which includes a problem solver and a user interface, while a FRS’s role is to be used as the starting point, or contract, for the software development process. Also, the requirements of a KBS are typically vague as it is hard to specify a KBS in advance.

- differing knowledge: a KBS often contains heuristic or probabilistic knowledge, which is built up incrementally over the system’s lifecycle.

On reflection these differences appear superficial: more recent research in expert systems has been focussed on second generation systems involving qualitative modelling of ‘deep’ knowledge. The creation of the KBS starts with the acquisition of a *domain model* which is *validated* and acts as a basis for system development; this is seen as providing the basis for a more robust KBS. As such it coincides with the domain model featured in Figure 1. In reference [3] the construction of this kind of domain model is considered essential for requirements re-use in software engineering. For a more detailed comparison of validation in software engineering and KBS, the reader can consult reference [21].

Interestingly, the state of the art in KBS validation as surveyed recently in reference [24] appears to resemble that of validation in FRS. The authors examine issues raised by the use of tools developed for automatic validation, pointing out that relatively little work has been done in automated validation of KBS when compared to that concerned with verification techniques. Verification, in KBS, is a kind of generalisation of the idea of discharging proof obligations on a formal specification that ensures self-consistency (for example proving operator satisfiability in VDM). It encompasses the idea of providing a separate body of constraints and proving that they logically follow from the knowledge base. Examples of these constraints are structural invariants that ensure the absence of redundant or looping rule sets.

Work in KBS validation now seems to be embracing integrated methods for validation, with environments that include verification and machine learning techniques [7, 25]. The aim of this work seems to be to provide a comprehensive, automated environment to promote quality by KB refinement. This development parallels that of our earlier work on the use of diverse validation of formalised requirements specifications outlined in the section above, and our ongoing research into the use of machine learning techniques discussed in section 4 below.

5 Using Machine Learning in Validation

5.1 Introduction

Much research in machine learning has been concerned with the acquisition and refinement of *Domain Theories* [10], using examples or counter examples as the main input to generalisation algorithms. Some work has concentrated on inducing theories ‘from scratch’ [15], whereas other work starts from the assumption that we have an imperfect theory already. Positive and negative instances are used for refinement of the imperfect theory [18]. Theories may be refined or adapted [2, 17] because they are inaccurate, or because they fail some operating criteria (as in the field of Explanation-based Learning [11]). These domain theories are typically written in a logic form, not dissimilar to that used for capturing requirements in the FAROAS project. Very often domain theories have been restricted to ‘toy’ domains, but it now seems feasible to use current techniques in automated refinement of first order logic theories modelling realistic applications [18]. ML also has been utilised to validate knowledge bases [6, 25] and in program development and verification [4]. In [5], a technique is described for extracting specifications by (machine) learning from example behaviours.

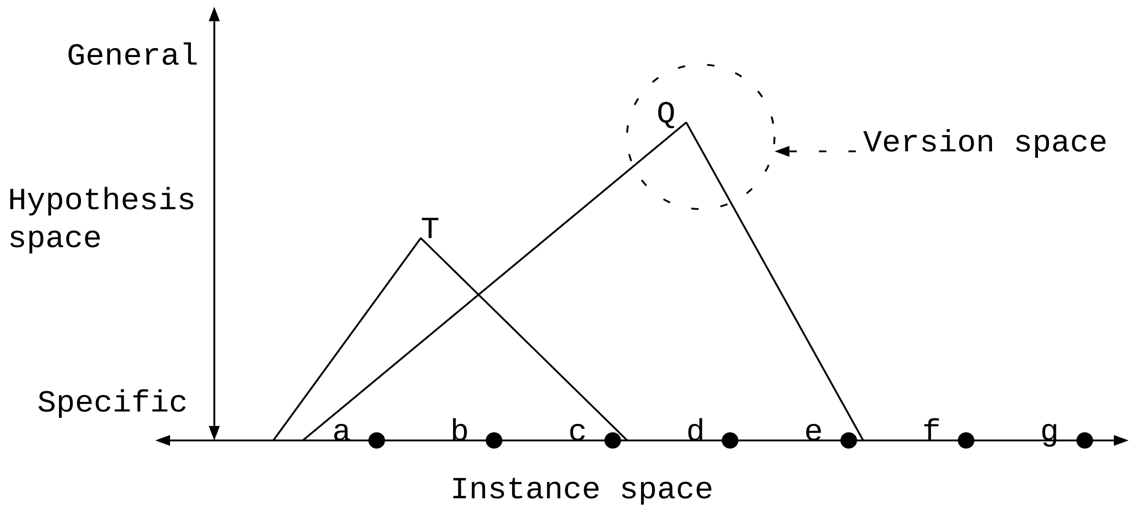

Figure 3: Hypothesis Space for Concept C

Our work is based on the premise that a body of formalised requirements can be viewed as an imperfect theory, and test cases are the examples and counter examples that the theory should conform to, driving a process of learning manifested as *theory revision*. A simplified but useful view of many ML processes including this one can be explained with the help of Figure 3. Assume the definition of a concept C is to be learned: the space above the horizontal axis is termed hypothesis space (HS), where points in HS are hypotheses of C's definition (i.e. *theories*), determined by a well defined hypothesis language. The horizontal axis represents instance space (IS), and points in IS can be positive or negative instances of C from which the desired hypothesis is induced. ML processes can often be modelled as search algorithms using *generalisation* operators to move up the space, and *specialisation* operators to move down. A useful partial order can be defined as follows:

For any Q and $T \in$ HS: $Q \succ T$ iff all the concept instances derivable using T can also be derived using Q ($\succ$ is read as 'covers').

In the case where $T \in$ IS, $Q \succ T$ iff T is derivable from Q. The subspace of HS that contains all valid hypotheses according to a set of positive instances P and negative instances N, is termed *version space* [14]. It can be defined as a set:

$$VS = \{Q \in \text{HS}: \forall p \in P, n \in N, Q \succ p \, and \, Q \not\succ n\}$$

The hypothesis language is important because it determines the space of possible definitions of C. Its choice introduces a *language bias*, where typically over-simplifications lead to a null VS, and over-complexity leads to an intractable search algorithm. In the field of theory revision, a further strong bias is introduced by starting with a point T in HS (a theory) which is near, but not in VS. Specialisation and generalisation operators are used to revise parts of T until a revision is found that is nearer VS i.e. it is closer to a complete coverage of the instances.

In Figure 3 the instances that a theory covers are represented at the base of the triangle below the theory. Let $P = \{a, b, c, d, e\}$ and $N = \{f, g\}$. T is an incorrect hypothesis for concept C if it only covers 3 of its positive instances. A successful theory revision algorithm would revise T to a theory Q which covered all of P and none of the N.

5.2 Applying Theory Revision to the CPS

We plan to use operators and tools such as those found in Richards and Mooney's FORTE system [18] to identify parts of the CPS in need of revision, and to automate the making of those revisions. Assume, therefore, that the CPS is an imperfect theory of conflict prediction. Using the model given in Figure 3:

The concept to be learned, *C* is the correct definition of the predicate

```
‘‘Profile1 and Profile2 are_in_oceanic_conflict’’
```

Hypothesis Space is the set of all theories allowable using the grammatical definition G(CPS) and that are in a *general clausal form*, headed by the conflict prediction predicate. This is essentially the same form as the executable output from the FREE’s translator.

The starting point for theory revision is the clausal form program version of the CPS.

Instance Space is a set of instances, marked positive or negative as appropriate, of the form:

```
‘‘X and Y are_in_oceanic_conflict’’
```

where profile identifiers X and Y are defined by a suitable set of flight information axioms.

Version Space is the subset of Hypothesis Space in which all positive instances are derivable from the members of the subset, and all negative instances are not.

The starting point for theory revision is the execution of a *blame assignment algorithm* which pinpoints which parts of the theory to revise. Proof trees of test runs which did not give the correct output (called ‘test failures’) are analysed, and those clauses which were used in these proofs are marked. Clauses which appear in the proof trees of the most test failures are passed on for possible revision.

A search through hypothesis space is performed by incrementally changing one or more of these (possibly) defective clauses. Given the complexity of a logic program generated from a specification such as the CPS, we need to make some assumptions to help to constrain search through hypothesis space:

- language bias: the existence of G(CPS) and the assumption that the CPS is in general clausal form;
- minimal revision: we assume the CPS is *nearly* correct, and only make minimal revisions to it;
- revisible theory restriction: parts of the CPS that are considered correct (for example, those that have been validated by other processes) can be kept immune or shielded from blame assignment.

For example, let us assume the blame algorithm outputs the following abstract clause for revision:

a(X,Y) :- b(X,Z), not(c(X)), d(Z,Y).

which was output from the blame assignment algorithm because it appeared in the proofs of test failures where negative instances were falsely derived. In the case study this would be equivalent to the CPS’s execution form deciding that two aircrafts’ profiles are in conflict, when in fact this is judged not to be the case by domain experts. The clause therefore needs to be specialised, and possible revision operators include:

(a) finding a predicate from the vocabulary of G(CPS) that has all its arguments of the same sorts as X, Z, Y, then adding this to the clause with $X, Z or Y$ as its arguments where appropriate;

(b) replacing ‘b’ (or ‘d’) with a predicate that is a specialisation of ‘b’ (or ‘d’), or ‘c’ with a predicate that is a generalisation of ‘c’. For example, the predicate ‘greater_than_or_equal_to’ may be specialised to ‘greater_than’;

(c) changing constants appearing in predicates by incrementing or decrementing their values.

A useful revision is one that rectifies the test failures and does not cause any of the proofs of successful tests to subsequently fail. To show how the CPS itself might be revised we will use two examples of bugs uncovered at the last validation meeting of the FAROAS project [12].

The first bug was found in an auxiliary axiom 2.1.4 defining whether segments fell within the oceanic boundaries of the separation criteria. The validation meeting spotted two errors, which (when viewed in executable form) changed the clause:

```
oceanic_separation_rules_are_applicable_to(Segment):-
    the_Profile_containing(Segment,Profile1),
    is_wholly_or_partly_in_shanwick_oca(Profile1),
    starts_at_or_after_first_recognised_pt_for_oceanic_cpr(Segment),
    ends_at_or_before_last_recognised_pt_for_oceanic_cpr(Segment),
    the_max_Flight_level_of_segment(Segment,Flight_level1),
    Flight_level1 is_at_or_above fl(55),
    not__((is_wholly_outside_shanwick_oca(Segment))).
```

to:

```
oceanic_separation_rules_are_applicable_to(Segment):-
    the_Profile_containing(Segment,Profile1),
    is_wholly_or_partly_in_oca(Profile1),
    starts_at_or_after_first_recognised_pt_for_oceanic_cpr(Segment),
    ends_at_or_before_last_recognised_pt_for_oceanic_cpr(Segment),
    the_max_Flight_level_of_segment(Segment,Flight_level1),
    Flight_level1 is_at_or_above fl(55),
    not__((is_wholly_outside_oca(Segment))).
```

The predicates affected are

```
is_wholly_or_partly_in_shanwick_oca
is_wholly_outside_shanwick_oca
```

Both have been replaced by more general versions of the predicates, is a manner similar (but opposite) to the revision operator (b) above.

The axiom 3.1.2, used as an example in section 3, was also subject to a revision at this validation meeting. The predicate:

```
(both Flight_level1 and Flight_level2 are_above FL 430)
```

was originally incorrectly stated as:

```
(both Flight_level1 and Flight_level2 are_above FL 450)
```

Using the revision operator (c) above this change is potentially within the capabilities of the revision tool. Although these examples show the theory reviser's potential, this is ultimately dependent on an adequate set of test cases being available.

5.3 The Augmented FREE

In Figure 2 the ML tool is shown as an integrated component of the FREE, for use in the validation and maintenance of a specification in customised MSFOL. Figure 4 outlines this subsystem in a little more detail. It has two processes:

- The Instance Classifier will take as input a special form of the executable called the Revisible Theory, a general clausal form translated from the CPS which is subject to incremental changes caused by theory revision. It will also input the Instances, which are composed of the test case queries and Profile axioms in clausal form. The process then outputs the Classified Instances, collecting test failures into groups of 'false positives' and 'false negatives' as predicted by the Revisible Theory.

- The Theory Reviser will take as input a classified set of instances and a Revisible Theory and output a new version of the Revisible Theory using some of the techniques discussed in the previous section. The new theory should increase coverage by classifying more instances correctly.

The revision process will iterate in a hill-climbing fashion until it reaches a new theory which correctly classifies all the positive and negative instances, or it reaches a local maximum. The final set of revisions output by the Theory Reviser will be fed back to the CPS via the modeller, who can then validate the proposed changes using any of the other processes 1 - 5 discussed in section 3.3.

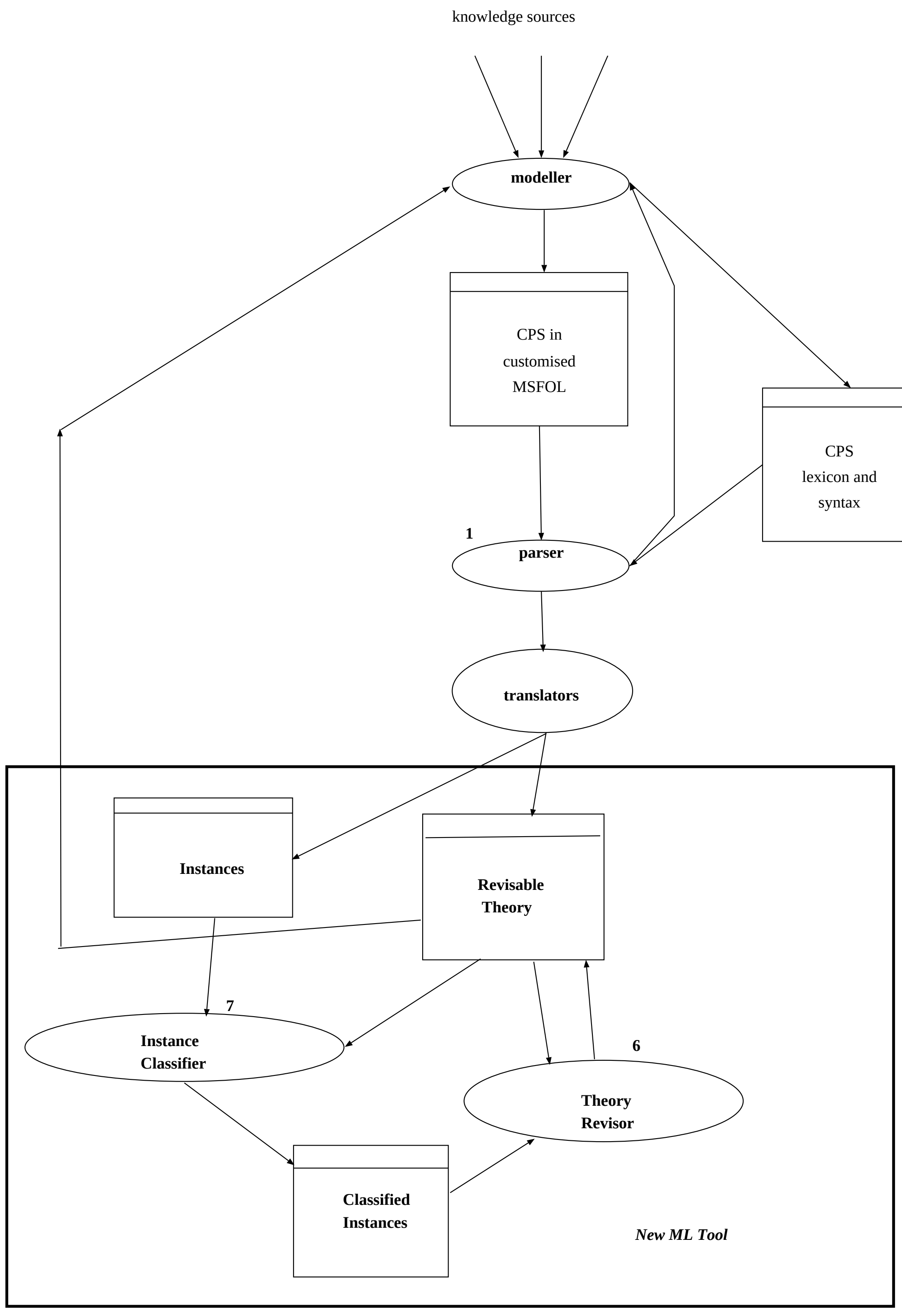

Figure 4: The New ML Tool in the FREE

There are two problems, however: (i) the complexity of the revision process (ii) adequacy of the customised language. The complexity issue is at least in part addressed by introducing strong biases into revisions as mentioned in section 5.2. Also, the ability to limit revisions to a subset of the theory's clauses can be used to cut down the search space. The adequacy issue is perhaps a more fundamental problem, as it is not uncommon to want to introduce new terms into the requirements language. Our future work will include looking into the reasons for a failure of the theory reviser to find a revised theory in version space. This at least could trigger the need to introduce new terms.

6 Summary

In this paper we reviewed some current approaches to the validation of formal specifications of requirements and described a case study which used a diverse validation environment to help debug requirements stated in a customised form of MSFOL. Although the combination of strategies proved particularly useful, automated processes such as testing which show the presence of bugs, fail to pinpoint or help repair them. After drawing an analogy with the validation of knowledge bases, we have argued for the use of machine learning techniques such as theory revision to augment a validation environment in order to help solve the problems of requirements specification refinement.

7 Acknowledgements

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

References

[1] Manual of Air Traffic Services Part 2 – Operations. Technical report, The Civil Aviation Authority – National Air Traffic Services.

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

[3] D. Bolton, S. Jones, D. Till, D. Furber, and S. Green. A Generic Modelling Approach to Requirements Capture in the Domain of Air Traffic Control. In *IEE Colloquium on "Software in ATC Systems – The Future"*, 1992.

[4] I. Bratko and M. Grobelnik. Inductive learning applied to program construction and verification. In *Proceedings of the ILP'93 Workshop*, Bled, Slovenia, April 1993.

[5] W.W. Cohen. Inductive specification recovery: Understanding software by learning from example behaviours. *Automated Software Engineering*, 1995.

[6] S. Craw, D. Sleeman, R. Boswell, and L. Carbonara. Is knowledge refinement different from theory revision? In S. Wrobel, editor, *MLnet Workshop on Theory Revision and Restructuring*, pages 33–35, 1994.

[7] S. Craw and D.H. Sleeman. Refinement in response to validation. *Expert Systems With Applications*, 8(3):343–349, 1995.

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

[9] 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.

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

[11] 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.

[12] T.L McCluskey. Report of validation meeting, oceanic area control centre conflict prediction and resolution study. Technical Report CAA/FAROAS, CAA/FAROAS/05/052/01, ISSUE: 01, December 1993.

[13] 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.

[14] T.M. Mitchell. *Version Space: An Approach to Concept Learning*. PhD thesis, Stanford University, Stanford, US, 1978.

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

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

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

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

[20] STARTS Public Purchaser Group. *The STARTS Purchasers' Handbook: Procuring Software-Based Systems*. NCC publications, UK, 1989.

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

[22] 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.

[23] B A Wichmann. A Development Model for Safety-Critical Software. In B A Wichmann, editor, *Software in Safety-Related Systems (IEE/BCS Joint Study Report)*, pages 211–223. John Wiley and Sons, UK, 1992. ISBN 0471-93474-7.

[24] N. Zlatareva and A. Preece. State-of-the-art in automated validation of knowledge-based systems. *Expert Systems with Applications*, 7(2):151–167, 1994.

[25] N.P. Zlatareva. A framework for verification, validation, and refinement of knowledge bases - the VVR system. *International Journal of Intelligent Systems*, 9(8):703–737, 1994.