Representational Reasoning and Verification

Gavin Doherty, José Campos and Michael D. Harrison

Human Computer Interaction Group,
University of York

Abstract. Formal approaches to the design of interactive systems, such as the *principled design* approach rely on reasoning about properties of the system at a very high level of abstraction. Such specifications typically provide little scope for reasoning about presentations and the *representation* of information in the presentation. Theories of distributed cognition place a strong emphasis on the role of representations in the cognitive process, but it is not clear how such theories can be applied to design.
In this paper we show how a formalisation can be used to encapsulate representational aspects, affording us an opportunity to integrate representational reasoning into the design process. We have shown in [3] how properties over the abstract state place requirements on the presentation if the properties are to be valid at the perceptual level, and we have presented a model for such properties. We base our approach on this model, and examine in more detail the issue of *verification*. Given the widespread consensus that proper tool support is a prerequisite for the adoption of formal techniques, we apply a higher-order logic theorem prover to the analysis.

1 Introduction

It has been proposed that formal techniques to modelling and specification can be used to improve the quality of interfaces to interactive systems. For example, the *principled design* approach aims to enhance the design process by ensuring conformance to certain *carefully chosen* design principles.

These principles, such as predictability, reactivity and support for the user's task attempt to relate the system specification to properties that have meaning for the user. This affords the designer the ability to reason about the usability of the system at a very early stage in the development life-cycle. Specifications which support such reasoning, for example those based on the interactor model [4], abstract away from the presentation as presentations typically include many details which are not relevant and are highly subject to change. Yet recent work on distributed and external cognition [10, 5, 12, 8], postulates that representations (both internal and external) play a critical role in the cognitive process. Hutchins [6], uses this distributed view to study the role emergent properties of a cockpit instrument play in helping the pilot perform his task. However, it is not clear how such analysis could be used to inform and improve the design process.

Additionally, reasoning such as that described above generates requirements over the presentation, which must hold if reasoning over the abstract specification is to be valid at the presentation level (ie. as the user perceives the system) [3].

1.1 A Formal Approach

The aims of this paper are twofold:

- to provide a rigorous and direct means for integrating representational reasoning in the style of [6] into the design process.
- to further explore issues of verification and show how the verification process exposes assumptions and requirements embedded in the presentation.

To address these issues we build upon the formalisation in [3]. The benefits of formality stem in this case from the ability to take a rigorous and methodical approach to a form of analysis which would otherwise be conducted in an ad-hoc fashion, and that we can apply the analysis to a *specification* of the system which has a formal relationship with the artefact. In [3] it is shown how representational requirements could be modelled in terms of a mapping between logical operators over the abstract state and perceptual operators over the presentation. Taking a formal approach to this involves constructing a model of the abstract artefact under consideration, its representation, and the mapping between them.

This is not to say that we advocate full formal specification of presentations, a proposition we see as neither practical nor valuable, but that a limited specification, including details of the *representation* and the operators supported by the representation is sufficient for establishing the validity of a presentation with respect to a property, such as support for a given task.

One of the emphases in this paper is on showing how the use of theorem proving can help in the automation of the verification process. In particular we will show how using a theorem prover forces us to *spell out* all the assumptions we are making about the system and its presentation, and identify problems with both the presentation and the system that should be behind it.

1.2 Overview

With this focus on machine-assisted verification in mind, we present in the next section a summary of the nature of the properties to be verified, along with a short example. Formalisation of this example reveals many representational issues. In the section following, we use this example in a proof using a higher-order logic theorem prover (PVS) to illustrate how the verification reveals further aspects of the presentation and the issues involved in providing machine assistance for the process. The formal notation employed is VDM-SL, but the approach can be applied to any model-based specification language.

2 A Model of Representation

We present in this section both a summary of the model in [3] and explore further the formalisation process.

2.1 Presentation Model

We begin with a simple functional model of the presentation mapping, which represents the abstract state (modelled as a set of attributes) to the presentation (modelled as a set of presentation elements, or *percepts*). A given piece of information may of course have many different possible representations.

1.0 $\rho : \mathrm{Attribute}\text{-set} \rightarrow \mathrm{Percept}\text{-set}$

Unlike the abstract state, the percepts represent information which is visible to the user, with no approximation or information loss. The mapping itself often (and necessarily) approximates the abstract attributes.

2.2 Logical and Perceptual Operators

Logical operators are those which may be defined over the abstract state (for example, magnitude comparison of two integers). Perceptual operators are those which may be defined over the presentation, and are understood to be directly performable by the user (eg. determining if two objects on a display are adjacent). In terms of the formalisation above, logical operators are defined over $\mathrm{Attribute}$-set, and perceptual operators are defined over $\mathrm{Percept}$-set. Both the logical and perceptual operators can hence be formalised.

The concepts of logical and perceptual operators have previously been applied by Casner [2] who constructed a system for automatic presentation generation by replacing logical operators in a task description by graphical components supporting perceptual versions of these operators. We take the converse view, and formulate our requirements on the presentation (for the specific example of task support), as follows:

> *to support a given task, the presentation should provide perceptual equivalents of the logical operators in the task*

This of course refers to the portion of the task to be performed by the user.

We believe that formalising the transformation from logical to perceptual operators provides an explicit and rigorous basis for reasoning about representational issues. For example, the scale types of Stevens [9], applied by Zhang [11] to the analysis of relational information displays, can be formalised in terms of the groups of logical and perceptual operators that each scale supports. In this way, we can use the operators to *characterise* the representation. By trying to formulate perceptual operators over the presentation model, we can expose hidden referents in our tasks. This process serves both to increase our understanding of the system, and acts as an aid to design.

2.3 A Model of Presentation Based Properties

One approach is to view the perceptual model as a reification of the abstract model, but proving the consistency of state changes between the abstract and reified models tells us nothing about how the presentation supports the desired properties.

For the reification to be valid, we must know that if a property holds on the abstract specification, it also holds on the presentation. We accomplish this by relating the abstract and perceptual models to a model in which the principle can be expressed. Establishing the validity of the presentation then becomes a matter of showing that the logical operators over the abstract state and the perceptual operators over the representation of this abstract state yield the same result in terms of the property (see figure 1). We can express this formally as:

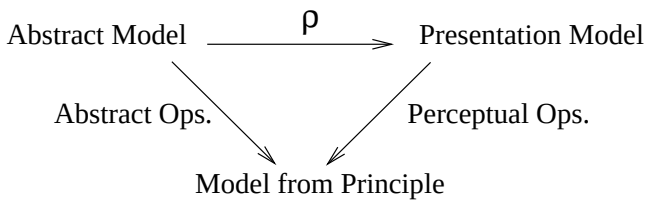

Fig. 1. Alternative approach to verification

$$2.0 \quad \text{Abstract-op}(\text{Attribute-set}) = \text{Perceptual-op}(\rho(\text{Attribute-set}))$$

2.4 An illustrative example

In this section, we present a formalisation of an application discussed by Hutchins [6]. Hutchins' approach involves a broader contextual view of the cockpit system, using understanding derived from 'distributed cognition'. By using this example, we hope to illustrate how our approach achieves a good coverage of the representational aspects of the analysis, and indicate how this might relate to a design context. The example concerns the use of 'speed bugs' to record minimum manoeuvering speeds on an aircraft air speed indicator.

The indicator takes the form of a circular scale on which a needle indicates the current air speed. The 'configuration change bugs' take the form of movable tabs on the perimeter of the instrument, and are set by the flight crew prior to the approach. The configurations referred to are the four configurations of wing slat and flap settings. One task of the crew is to change between them to generate more lift as the aircraft slows for landing. In addition, there is also a 'salmon bug', internal to the instrument, which denotes the speed commanded to the auto-throttle system. In the final approach they must also ensure the aircraft is within a safe limit of the reference speed (which should be that commanded to the auto-throttle system).

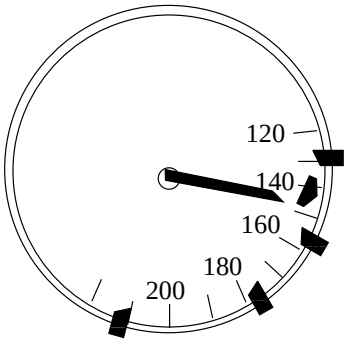

Fig. 2. Simplified Air Speed Indicator, adapted from [6]

Logical Model What is interesting about this artefact and the tasks it supports is that although on the surface it appears simple, there are in fact many pieces of information required to perform the operations involved in the tasks. This becomes apparent when we formulate an initial description of the configuration management task:

if *current speed* is within *acceptable margin speed*
of *minimum manoeuvering speed* for *current configuration*
then change to *next configuration*

We can see from this that there are four referents in the operation of changing configuration: the current speed, the margin speed, the minimum manoeuvering speed and the configurations being changed between (treated as one since they are paired). A logical operator to support this task could be one which checks a speed and configuration and determines whether it is appropriate to change to the next configuration. The task of tracking potential disparity between the indicated air speed and reference speed on the approach could be supported by a logical operator which takes a speed and returns whether it a safe deviation from $\mathrm{V_{ref}}$.

The first step in our formalisation is to construct a model of the ASI:

values

3.0 $\mathrm{S_{margin}} : \mathrm{Speed} = 10$ KNOTS — Margin for configuration change ;

4.0 $\mathrm{S_{safe}} : \mathrm{Speed} = 5$ KNOTS — Variation from approach speed

types

5.0 $\mathrm{Speed} = \mathbb{R}$;

6.0 $\mathrm{Configuration} = \mathbb{N}_1$;

7.0 $\mathrm{AbstractASI} :: \mathrm{V_c} : \mathrm{Speed}$ — Current Speed
.1 $\mathrm{C_c} : \mathrm{Configuration}$ — Current configuration
.2 $\mathrm{S_{mm}} : \mathrm{Speed}^+$ — Minimum manoeuvering speeds
.3 $\mathrm{V_{ref}} : \mathrm{Speed}$ — Reference speed of approach

We can now formalise the logical operation to support the configuration management task as:

8.0 $\mathrm{configChangeCheck} : \mathrm{AbstractASI} \to \mathbb{B}$
.1 $\mathrm{configChangeCheck}\,(\mathrm{asi}) \triangleq$
.2 $\quad \mathrm{asi.V_c} \leq \mathrm{asi.S_{mm}}(\mathrm{C_c}) + \mathrm{S_{margin}}$

The operation for checking the approach speed is as follows:

9.0 $\mathrm{approachSpeedCheck} : \mathrm{Speed} \times \mathrm{Speed} \to \mathbb{B}$
.1 $\mathrm{approachSpeedCheck}\,(\mathrm{V_c}, \mathrm{V_{ref}}) \triangleq$
.2 $\quad \mathrm{V_{ref}} - \mathrm{S_{safe}} \leq \mathrm{V_c} \leq \mathrm{V_{ref}} + \mathrm{S_{safe}}$

We can see that this formal model provides a concise 'computational' view of the operations, and the information required to carry them out.

Presentation Model The two main percepts are the ASI needle and the speed bugs. The scale is also a percept, but we use it only to establish a relationship between angles and sections of arc on the display, and absolute speeds.

We begin the specification with the data types representing the percepts. Perceptually the ASI needle is simply an angle from the upright ($0\,\mathrm{deg}$) position. A speed bug has both a position (again, an angle from the upright), and an extent, an angle which describes an arc to either side of the position. The perceptual function of the scale is to relate angles on the display to speeds.

values

10.0 $\mathrm{ScaleFactor} : \mathbb{R} = 1$ — Unit speed per scale degree ;

11.0 $\mathrm{SalmonExtent} : \mathrm{Angle}$ is not yet defined;

12.0 $\mathrm{ConfigExtent} : \mathrm{Angle}$ is not yet defined

types

13.0 $\mathrm{Angle} = \mathbb{R}$;

14.0 $\mathrm{ASINeedle} :: \mathrm{posn} : \mathrm{Angle}$;

15.0 $\mathrm{ASISpeedBug} :: \mathrm{posn} : \mathrm{Angle}$
.1 $\qquad\qquad\qquad\quad \mathrm{extent} : \mathrm{Angle}$;

16.0 $\mathrm{ASIScale} :: \mathrm{interpret} : \mathrm{Angle} \to \mathbb{R}$

The full instrument integrates these three components, note that we have a *sequence* of speed bugs, arranged in order of decreasing angle (and hence represented speed):

17.0 $\text{ASI\_Instrument} :: \text{needle} : \text{ASINeedle}$
.1 $\text{bugs} : \text{ASISpeedBug}^+$
.2 $\text{salmonbug} : \text{ASISpeedBug}$
.3 $\text{scale} : \text{ASIScale}$

.4 **inv** $\text{asi} \triangleq$ — Speed bugs cannot overlap
.5 $\forall \text{i}, \text{j} \in \textsf{dom}\ \text{bugs} \cdot$
.6 $\text{i} < \text{j} \Rightarrow \text{asi.bugs(i).posn} > \text{asi.bugs(j).posn}$
.7 $\wedge\ \text{asi.bugs(i).posn} - \text{asi.bugs(i).extent}$
.8 $> \text{asi.bugs(j).posn} + \text{asi.bugs(j).extent}$

We are now in a position to formalise the presentation mapping between these two models. The presentation ρ, maps an AbstractASI value to an ASI_Instrument value which represents it.

18.0 $\rho : \text{AbstractASI} \to \text{ASI\_Instrument}$
.1 $\rho\,(\text{a}) \triangleq$
.2 $\text{mk-ASI\_Instrument}\,(\rho\text{-Needle}(\text{a.V}_\text{c}), \rho\text{-BugSeq}(\text{a.S}_\text{mm}),$
.3 $\rho\text{-Salmon}(\text{a.V}_\text{ref}), \rho\text{-Scale})$

19.0 $\rho\text{-Needle}\,(\text{v}) \triangleq$
.1 $\text{mk-ASINeedle}\,(\text{v}/\text{ScaleFactor})$

20.0 $\rho\text{-BugSeq}\,(\text{s} : \text{Speed}^+)\ \text{bs} : \text{ASISpeedBug}^+$
.1 **post** $\forall \text{i} \in \textsf{dom}\ \text{s} \cdot \text{bs(i)} = \text{mk-ASISpeedBug}\,(\text{s(i)}/\text{ScaleFactor},$
.2 $\text{ConfigExtent})$
.3 $\wedge\ \text{len(s)} = \text{len(bs)}$

21.0 $\rho\text{-Salmon}\,(\text{v}) \triangleq$
.1 $\text{mk-ASISpeedBug}\,(\text{v}/\text{ScaleFactor}, \text{SalmonExtent})$

22.0 $\rho\text{-Scale}\,() \triangleq$
.1 $\text{mk-ASIScale}\,(\lambda\,\text{a} : \text{Angle} \cdot \text{a} * \text{ScaleFactor})$

Perceptual Operators The first perceptual operator we examine is that to support the task of checking the approach speed (represented by needle) against the reference speed (represented by the salmon bug). As explained by Hutchins [6], the operation reduces to checking whether the ASI needle position falls within the section of arc covered by the speed bug.

23.0 $\text{salmonBugCheck}\,(\text{needle}, \text{bug}) \triangleq$
.1 $\text{in\_arc}\,(\text{needle}, \text{bug.posn} - \text{bug.extent}, \text{bug.posn} + \text{bug.extent})$

24.0 $\text{in\_arc}\,(\text{needle}, \text{a}_{\text{start}}, \text{a}_{\text{end}}) \triangleq$

.1 $\text{a}_{\text{start}} \leq \text{needle.posn} \leq \text{a}_{\text{end}}$

Part of what makes this artefact so effective from a representational point of view is the simplicity of the perceptual operators for tasks such as this.

Moving on to support for the configuration management task, firstly there is the issue of comparing the current speed to the configuration change bugs. This is more complex than for the salmon bug, as the spatial extent of the bugs does not correspond to the margin within which it is appropriate to change. Also, it is asymmetric, since although it is acceptable to make changes within a certain margin above the minimum manoeuvering speed, it is not acceptable to go below this speed before making the configuration change.

Thus the perceptual operation must be one which determines (one sided) proximity of the ASI needle to the speed bug, relative to a margin which is *not represented in the presentation*.

25.0 $\text{configBugCheck}\,(\text{needle}, \text{bug}) \triangleq$

.1 $\text{in\_arc}\,(\text{needle}, \text{bug.posn}, \text{bug.posn} + (\text{S}_{\text{margin}}/\text{ScaleFactor}))$

The presence of an element of the abstract state (S_{margin}) indicates a hidden referent in the operation. This does not necessarily indicate a serious inadequacy of the presentation, (for example, information may sometimes be provided by other artifacts), although it does point to a lack of integration with the other percepts involved in the operation. In this case S_{margin} is a constant which is not as critical as deviation from the approach speed (Hutchins describes this task as being constrained by "Operational considerations" [6]). Thus, while this requires the pilot to have some internal representation of an acceptable margin for the configuration change, the configuration change task is not as heavily loaded as the approach speed task and so may be an acceptable cognitive burden on the pilot.

But there is also the issue of the current and next configuration. Ultimately, the perceptual operator must relate the abstract artefact (a sequence of minimum manoeuvering speeds) to a sequence of speed bugs around the perimeter of the ASI. We could use the ordering of the bugs as a simple formalisation, thus we employ a perceptual operator which indexes the sequence of speed bugs with the current configuration:

26.0 $\text{getCurrentBug}\,(\text{C}_{\text{c}}, \text{bugs}) \triangleq$

.1 $\text{index}\,(\text{C}_{\text{c}}, \text{bugs})$

We can see in this expression a requirement that the user already know the current configuration (C_{c}) or that it be represented in another artefact (which itself must enable the user to extract the information perceptually). Another (and perhaps more realistic) formalisation would be based on proximity of the ASI needle. Thus the 'next lowest' speed bug on the ASI is the one we want.

27.0 $\text{getCurrentBug}\,(\text{needle}, \text{bugs}) \triangleq$

.1 $\text{next\_counterclockwise}\,(\text{needle}, \text{bugs})$

28.0 next_counterclockwise (needle: ASINeedle, bugs: ASISpeedBug^+) bug: ASISpeedBug

.1 **pre** $\exists\, i \in \mathbb{N} \cdot \text{bugs}(i).\text{posn} < \text{needle.posn}$

.2 **post** $\exists\, i \in \mathbb{N} \cdot \text{bugs}(i) = \text{bug} \wedge \text{bugs}(i).\text{posn} \leq \text{needle.posn}$

.3 $\qquad \wedge\, \forall\, j \in \mathbb{N} \cdot j < i \;\Rightarrow\; \text{bugs}(j).\text{posn} > \text{needle.posn}$

Integrating these two operations into a composite operation to support the configuration change task yields:

29.0 $\text{asiConfigCheck} : \text{ASI\_Instrument} \rightarrow \mathbb{B}$

.1 $\text{asiConfigCheck}\,(\text{asi}) \triangleq$

.2 $\quad \text{configBugCheck}\,(\text{asi.needle}, \text{getCurrentBug}\,(\text{asi.needle}, \text{asi.bugs}))$

We can see from the above that the process of formalisation itself contributes to our understanding of the system, and possible shortcomings. The next section will illustrate how more complex assumptions embedded in the representation may be discovered.

3 Verification

Having defined both the abstract and presentation models, operators over these models and the mapping between them, we can proceed with the verification phase of our analysis. Recall that the form of verification we are using involves establishing an equivalence between logical operators over the abstract state and perceptual operators over the presentation.

3.1 The prover - PVS

In this section we will introduce PVS, the theorem prover that will be used in the verification process that follows. Our aim is to enable the reader to follow the subsequent description of the performed verification. See [7] for a more thorough introduction to the system.

PVS is a typed higher-order logic theorem prover, which provides an integrated environment for development and analysis of specifications. Specifications are organised in theories. Typically a theory will introduce a number of types and constants (which can be functions), and formulas associated with them (axioms and theorems, for instance). Theories can be parameterised on types and constants. Entities declared in a theory can be made available to others by exporting them (using the EXPORTING clause). By default all declarations are exported. Entities that are exported in a theory can be imported by another using the IMPORTING clause.

PVS features a powerful type system. This is very useful when writing specifications, but means that type checking becomes undecidable. To cope with this, the type checker generates proof obligations (TCCs - Type Correctness Conditions) that must be

established by the theorem prover. If the system is unable to prove a TCC automatically, then the user is asked to do it.

The usual types are available in PVS: natural numbers (nat), real numbers (real), sequences ($\mathrm{sequence}[\mathcal{X}]$), sets ($\mathrm{set}[\mathcal{X}]$), tuples ($[\#...\#]$), etc. These types are either built-in in the system or defined in the prelude library. PVS also allows for the definition of predicate subtypes, dependent types and abstract data types. Although we will not use these directly, they are used in the prelude to define some of the types we will be using.

A library is a collection of theorems. The prelude is a special library whose theories are always available, without the need for explicit importing.

PVS is used interactively. Its interface is mainly implemented as an Emacs major mode, which integrates functionality for editing specifications and proving theorems. When performing a proof, the system presents a goal in the form of a sequent, and prompts the user for an appropriate command. If the command does not solve the sequent, it will generate a new sequent or a number of new sequents (ie. subgoals), and the user will be asked to prove them in turn. When all subgoals are proved the original goal has been proved.

The user interacts with the prover by issuing commands to be applied to the sequent. There are commands for induction, quantifier reasoning, rewriting, simplification using decision procedures and type information, and propositional simplification using binary decision diagrams.

3.2 Writing the specification in PVS

The translation from VDM to PVS is straightforward [1]. The specification is organised into three theories. One for the logical model (theory ASI), another for the perceptual model (theory $\mathrm{perceptualASI}$), and a final theory that combines the previous ones and introduces the equivalences to be proved (theory $\mathrm{ASIverification}$).

Figure 3 presents the PVS theory for the logical model as described in section 2.4. The theory starts by introducing the four types. Speeds is declared as a sequence of Speed, and $\mathrm{abstractASI}$ as a tuple with four elements (Vc, Cc, Smm, and Vref). After the types, two constants are introduced — $\mathrm{Smargin}$ and Ssafe — both of type Speed. Note that they are left uninterpreted (ie. no actual values are given). The variable $\mathrm{abs\_asi}$ is declared and used in the axiom that follows (axiom $\mathrm{inv\_abs\_asi2}$). This axiom asserts the invariant of $\mathrm{abstractASI}$. Finally, the theory declares the two logical operators.

The theory in figure 3 was obtained by translating the VDM specification. As will be shown, the verification process will prompt us to introduce some changes in the specifications. In Appendix A we present the final ASI theory that resulted from the verification process. The $\mathrm{perceptualASI}$ theory is defined similarly. The final version (after verification) is given in Appendix B.

Note that we have added an axiom ($\mathrm{bug\_posn\_ext}$), expressing an invariant condition needed for the $\mathrm{ASISpeedBug}$ type theory, which was not included in the VDM specification. This restriction was identified by PVS as a TCC for $\mathrm{ASISpeedBug}$. The last theory, $\mathrm{ASIverification}$ (see figure 4), imports the two previous theories and introduces the equivalences to be proved as conjectures (ie. putative theorems).

```
ASI : THEORY
  BEGIN

  Speed : TYPE = nat
  Speeds : TYPE = sequence[Speed]
  Configuration : TYPE = nat
  AbstractASI : TYPE = [# Vc : Speed, Cc : Configuration, Smm : Speeds, Vref : Speed#]

  Smargin : Speed
  Ssafe : Speed

  abs_asi : VAR AbstractASI
  inv_abs_asi2 : AXIOM ∀ (i, j : nat) : i < j ⇔ Smm(abs_asi)(i) > Smm(abs_asi)(j)

  configChangeCheck((asi : AbstractASI)) : bool = Vc(asi) ≤ Smm(asi)(Cc(asi)) + Smargin
  approachSpeedCheck((Vc, Vref : Speed)) : bool = Vref - Ssafe ≤ Vc ∧ Vc ≤ Vref + Ssafe

  END ASI
```

Fig. 3. Initial version of the logical model

```
ASIverification : THEORY
  BEGIN

  IMPORTING ASI, perceptualASI

  vc, vref : VAR Speed

  approach_speed_task : CONJECTURE
    approachSpeedCheck(vc, vref) = salmonBugCheck(rho_Needle(vc), rho_Salmon(vref))

  abs_asi : VAR AbstractASI

  configuration_change_task : CONJECTURE
    configChangeCheck(abs_asi) = asiConfigCheck(ρ(abs_asi))

  END ASIverification
```

Fig. 4. ASIverification theory

3.3 Approach speed task

The first conjecture ($\mathrm{approach\_speed\_task}$) declares the equivalence between the operators $\mathrm{approachSpeedCheck}$ and $\mathrm{salmonBugCheck}$. When we start the proof attempt for this conjecture we get the sequent:

Sequent 1 *approach_speed_task:*

$$\vdash \{1\}\ (\forall\ (\mathrm{vc}:\ \mathrm{Speed}, \mathrm{vref}:\ \mathrm{Speed}):\ \mathrm{approachSpeedCheck}(\mathrm{vc}, \mathrm{vref}) = \mathrm{salmonBugCheck}(\mathrm{rho\_Needle}(\mathrm{vc}), \mathrm{rho\_Salmon}(\mathrm{vref})))$$

If we apply a `grind` (a powerful, somewhat brute force, rule which, among other commands, applies rewriting and simplification) we are left with four subgoals to prove. The sequent for the first subgoal is (where priming is used to indicate skolem constants[1]):

Sequent 2 *approach_speed_task.1:*

$$\{\text{-}1\}\ \mathrm{vc}' \geq 0,\ \{\text{-}2\}\ \mathrm{vref}' \geq 0,\ \{\text{-}3\}\ \mathrm{vref}' - \mathrm{Ssafe} \leq \mathrm{vc}',\ \{\text{-}4\}\ \mathrm{vc}' \leq \mathrm{Ssafe} + \mathrm{vref}' \vdash \{1\}\ (\mathrm{vc}' / \mathrm{ScaleFactor} \leq \mathrm{SalmonExtent} + \mathrm{vref}' / \mathrm{ScaleFactor})$$

What PVS is asking us to prove is that, if the current velocity (vc') is inside the safe margin of the reference speed (vref'), then the needle must be below the speed indicated by the edge of the salmon bug when positioned at vref' in the dial (consequent 1).

Assumption 1 The need to prove this derives from our assumption, at the perceptual level, that the extent of the salmon bug indicates the safe margin (a logical referent) of the reference speed. Hence we need to introduce a formal relation between the extent of the bug and the safe margin speed - a constraint spanning the abstract and perceptual levels. This is done by defining $\mathrm{SalmonExtent}$ as an interpreted constant which represents, at the perceptual level, the value of Ssafe:

$$\mathrm{SalmonExtent}:\ \mathrm{Angle} = \mathrm{Ssafe} / \mathrm{ScaleFactor}$$

Once this is done it becomes easy to prove sequent 2. The other three sequents are of similar nature and easily proven once the relation above is established. The proof tree for the final proof is shown in figure 5. The rewriting of $\mathrm{SalmonExtent}$ to $\mathrm{Ssafe}/\mathrm{ScaleFactor}$ is not shown explicitly as it is done automatically by the `grind` rule.

[1] Skolem constants are arbitrary representatives for quantified variables.

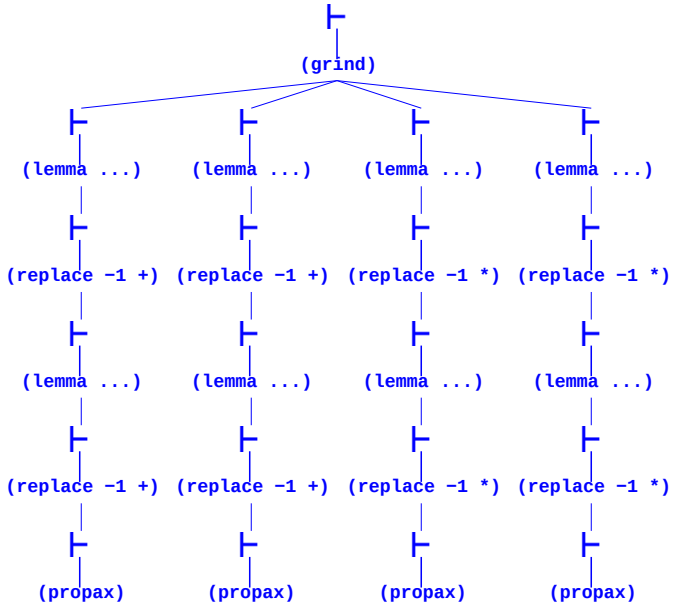

Fig. 5. Proof tree for approach_speed_task

3.4 Configuration change task

Having proved the equivalence of the logical and perceptual operators of the approach speed task in the revised version of the specification, we will now attempt to do the same for the configuration change task. The conjectures that represent the equivalence are introduced in theory ASIequivalence (see figure 4). The first sequent for this conjecture is:

Sequent 3 *configuration_change_task:*

$$\{1\}\ (\forall\ (\text{abs\_asi} :\ \text{AbstractASI}):\ \text{configChangeCheck}(\text{abs\_asi}) = \text{asiConfigCheck}(\rho(\text{abs\_asi})))$$

We start the proof by skolemizing and expanding definitions. Eventually we reach a point where, after introducing the definition of next_counterclockwise, the proof splits into two subgoals (*configuration_change_task.1* and *configuration_change_task.2*).

Proceeding with the first subgoal, after expanding definitions and some arithmetic simplifications the proof is again divided into two further subgoals (*configuration_change_task.1.1* and *configuration_change_task.1.2*). The first is represented by the following sequent, where ccbugindex is a skolem constant representing the index of the first bug just below the needle (which we interpret in configChangeCheck as representing the current configuration):

Sequent 4 *configuration_change_task.1:*

$$
\begin{array}{l}
\{\text{-1}\}\ \mathrm{posn}(\mathrm{rho\_BugSeq}(\mathrm{Smm}(\mathrm{abs\_asi}'))(\mathrm{ccbugindex})) \leq \\
\quad \mathrm{posn}(\mathrm{rho\_Needle}(\mathrm{Vc}(\mathrm{abs\_asi}'))) \\
\{\text{-2}\}\ (\forall\ (\mathrm{j}:\ \mathrm{nat}): \\
\quad \mathrm{j} < \mathrm{ccbugindex} \Rightarrow \\
\quad\ \mathrm{posn}(\mathrm{rho\_BugSeq}(\mathrm{Smm}(\mathrm{abs\_asi}'))(\mathrm{j})) \\
\quad\quad > \mathrm{posn}(\mathrm{rho\_Needle}(\mathrm{Vc}(\mathrm{abs\_asi}')))) \\
\hline
\{1\}\ (\mathrm{Vc}(\mathrm{abs\_asi}') \leq \mathrm{Smm}(\mathrm{abs\_asi}')(\mathrm{Cc}(\mathrm{abs\_asi}')) + \mathrm{Smargin} = \\
\quad \mathrm{configBugCheck}(\mathrm{rho\_Needle}(\mathrm{Vc}(\mathrm{abs\_asi}')), \\
\quad\ \mathrm{rho\_BugSeq}(\mathrm{Smm}(\mathrm{abs\_asi}'))(\mathrm{ccbugindex})))
\end{array}
$$

The sequent can be read as: if needle points to a velocity above or equal to the bug ccbugindex (antecedent -1), and all bugs above ccbugindex are also above the needle (antecedent -2), then, testing that the current (abstract) velocity $Vc(abs\_asi')$ is below the minimum manoeuvering speed of the current configuration $Smm(abs\_asi')(Cc(abs\_asi'))$ plus the safe margin, yields the same result as performing a configBugCheck on the needle and the bug just below the needle.

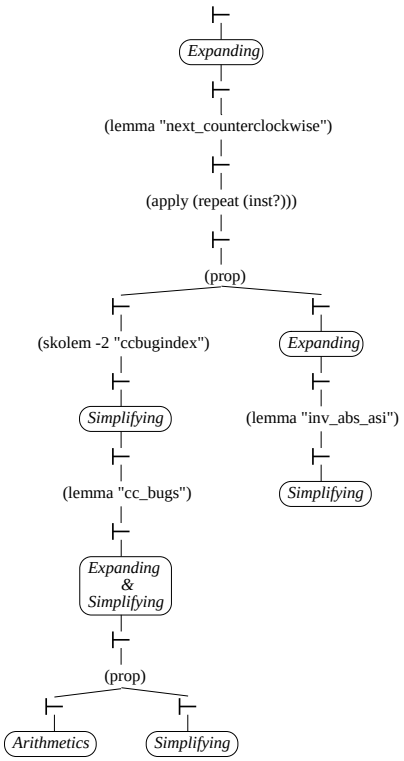

Fig. 6. Schematic proof tree for configuration_change_task

This is not unexpected and should be true, as we interpret the bug just below the needle as indicating the current configuration. To prove it, we must be able to establish that $\mathrm{Cc}(\mathrm{abs\_asi}')$ (the configuration index at the logical level) and $\mathrm{ccbugindex}$ (the configuration index at the perceptual level) point to the same minimum manoeuvering speed.

Assumption 2 Thus we expose the second assumption - that the bug just below the needle indicates the current configuration. The effect of this assumption is to assign a meaning (in terms of configuration) to regions of the airspeed indicator face - a vital representational property that Hutchins dwells on in some detail [6]. To proceed with the proof, we formalise the assumption with the following axiom:

```
cc_bugs : AXIOM
  ∀ (i : nat):
   ((posn(bugs(asi)(i)) × ScaleFactor ≤ Vc(abs_asi))∧
     (¬∃ (j : nat):
        j < i∧
          (posn(bugs(asi)(j)) × ScaleFactor ≤
            Vc(abs_asi)))) ⇒
    i = Cc(abs_asi)
```

After applying the axiom, we proceed once again by expanding definitions and simplifying. Eventually, the subgoal is further subdivided into two subgoals, both of which are easy to prove. We are left with subgoal *configuration_change_task.2*. This subgoal is represented by the sequent:

Sequent 5 *configuration_change_task.2:*

```
|
|-------------------------------------------------------------
|{1} (∃ (i : nat):
|       (Smm(abs_asi')(i) / ScaleFactor ≤ Vc(abs_asi') / ScaleFactor))
```

Assumption 3 This condition is generated by the precondition to *next_counterclockwise*. Combining this with the axiom *cc_bugs* yields a third assumption about the system. As failing to change configuration before dropping below the minimum manoeuvering is a critical item, it is generally assumed that this is never the case. Without knowledge of this constraint, we would have to conclude that the presentation is inadequate for representing configuration and would have to be changed or used in conjunction with some other perceptual artefact. Formalising the assumption, we introduce the following invariant for ASI:

```
inv_abs_asi : AXIOM ∀(abs_asi : ASI) : Vc(abs_asi) ≥ Smm(abs_asi)(Cc(abs_asi))
```

A very interesting aspect of this assumption is that it is an *operational* constraint (that the pilot *must* keep the aircraft in an appropriate configuration for the speed), on which the success of the representation (that it reflects the current configuration) is based, and further illustrates how abstract and representational properties are intertwined. Using the above invariant the proof for this subgoal can be finished. The proof tree for the final proof is shown in figure 6.

3.5 Summary

In the process of verification, we identified three significant changes, in the form of assumptions about the system, to the specification. Each of these assumptions is derived as a direct result of the sequent at which it is introduced. Two of these highlighted aspects of the abstract level that needed to be better represented at the perceptual level:

- the extent of the salmon bug had to be explicitly related to the safe speed margin, in order for the perceptual operator $\mathrm{configBugCheck}$ to work properly;
- the relationship between bugs and current configuration had to be made explicit in order for both operators of the configuration change task to be equivalent.

The third brought out the implications of some of the assumptions we were making about the interface and showed us that those issues were not been included in our abstract model. The most important point to grasp is that each of the assumptions added to enable completion of the proof had a representational basis and concerned vital properties of the presentation.

In summary, checking for consistency between an abstract specification of the interactive system and the presentation for that system, allowed us not only to better understand the issues involved at the interface level and what assumptions were being made, but also to see how assumptions made at the perceptual level relate back to the system itself. Additionally, the verification process was valuable in that it allowed us to identify a number of minor bugs in both specifications.

Finally, although these proofs are not so complex that it would be infeasible do them by hand, we found the prover to be helpful in two ways:

- PVS has a number of powerful proving commands that, most of the time, can save us a lot of time and patience — as we use more and more concrete specifications of the perceptual level, so the theorem prover will become more and more useful in this regard
- at a different level, by being *totally impartial*, the theorem prover better exposes assumptions we are making about the system. Had the verification been performed by hand, some of those assumptions might have crept into the proof unnoticed.

4 Conclusions

The stated aims of this paper were to provide a means for integrating representational reasoning into a design process, and to explore further the verification process.

We have shown that by employing a *formal* model which allows us to address representational issues, we provide both a rigorous and precise framework for reasoning about representation, and confidence that reasoning over the abstract state holds at the presentation level.

We have also shown how the verification process improves our understanding of the specification, and in particular brings out assumptions about the system which are embedded in the representation. We consider it an interesting aspect of the process that some issues have emerged purely from the effort of formalisation, whereas others emerge only when we attempt to verify the relationship between the logical and perceptual operators.

Each of the assumptions brought out by the verification process had an important representational significance and a direct correspondence with Hutchins analysis. While the first concerned a simple relationship between abstract state and representation, the second established the *implicit* representation of current configuration by the configuration change bugs. The final assumption exposed an operational constraint, which is not part of the abstract system model, yet which is vital for the success of the presentation. Thus it is not merely a question of errors or omissions in the specification, but additional information and understanding which emerges from the analysis.

Representations are, of course, only one part of the distributed cognitive system; one interesting area for future work would be to consider a wider range of cognitive resources, for example by employing the resources model of [10].

References

1. S. Agerholm. Translating specifications in VDM-SL to PVS. In *Proceedings of 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, 1996.
2. S.M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, April 1991.
3. G. Doherty and M. D. Harrison. A representational approach to the specification of presentations. In M.D. Harrison and J.C. Torres, editors, *Proceedings, 4th Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, Springer Computer Science. Springer Wien, 1997.
4. D. J. Duke and M.D. Harrison. Abstract interaction objects. *Proceedings of Eurographics '93*, Computer Graphics Forum, 12(3), 1993.
5. E. Hutchins. *Cognition in the Wild*. MIT Press, 1995.
6. E. Hutchins. How a cockpit remembers its speed. *Cognitive Science*, 19:265–288, 1995.
7. S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI Internatinal, Menlo Park CA 94025, USA, (beta release) edition, March 1993.

8. M. Scaife and Y. Rogers. External cognition: how do graphical representations work? *International Journal of Human-Computer Studies*, 45:185–213, 1996.
9. S.S. Stevens. On the theory of scales of measurement. *Science*, 103:677–680, 1946.
10. P.C. Wright, B. Fields, and M.D. Harrison. Distributed information resources: An new approach to interaction modelling. In T.R.G. Green, J.J. Canas, and C.P. Warren, editors, *Proceedings of ECCE8: European Conference on Cognitive Ergonomics*, pages 5–10. EACE, 1996.
11. J. Zhang. A representational analysis of relational information displays. *International journal of human computer studies*, 45, 1996.
12. J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.

A ASI PVS theory

```
ASI : THEORY
  BEGIN

  Speed : TYPE = nat

  Speeds : TYPE = sequence[Speed]

  Configuration : TYPE = nat

  AbstractASI : TYPE = [# Vc : Speed, Cc : Configuration, Smm : Speeds, Vref : Speed#]

  Smargin : Speed

  Ssafe : Speed

  abs_asi : VAR AbstractASI

  inv_abs_asi : AXIOM Vc(abs_asi) ≥ Smm(abs_asi)(Cc(abs_asi))

  inv_abs_asi2 : AXIOM ∀ (i, j : nat) : i < j ⇔ Smm(abs_asi)(i) > Smm(abs_asi)(j)

  configChangeCheck((asi : AbstractASI)) : bool = Vc(asi) ≤ Smm(asi)(Cc(asi)) + Smargin

  approachSpeedCheck((Vc, Vref : Speed)) : bool = Vref - Ssafe ≤ Vc ∧ Vc ≤ Vref + Ssafe

  END ASI
```

B perceptualASI PVS theory

```
perceptualASI : THEORY
  BEGIN
```

```
ASSUMING

  IMPORTING ASI

  ENDASSUMING

Angle : TYPE = nonneg_real

ASINeedle : TYPE = [# posn : Angle#]

ASISpeedBug : TYPE = [# posn : Angle, extent : Angle#]

ASISpeedBugs : TYPE = sequence[ASISpeedBug]

ASIScale : TYPE = [# interpret : [Angle → nonneg_real] #]

ASIInstrument:
  TYPE = [# needle : ASINeedle, bugs : ASISpeedBugs, salmonbug : ASISpeedBug, scale : ASIScale#]

ScaleFactor : posreal

BugExtent : Angle

SalmonExtent : Angle = Ssafe / ScaleFactor

bug_posn_extent : AXIOM ∀ (bug : ASISpeedBug) : posn(bug) - extent(bug) ≥ 0

asi : VAR ASIInstrument

abs_asi : VAR AbstractASI

inv_ASIInstrument : AXIOM
    ∀ (i, j : nat):
      i < j ⇒
        (posn(bugs(asi)(i)) > posn(bugs(asi)(j))∧
          posn(bugs(asi)(i)) - extent(bugs(asi)(i))
                > posn(bugs(asi)(j)) + extent(bugs(asi)(j)))

cc_bugs : AXIOM
    ∀ (i : nat):
      ((posn(bugs(asi)(i)) × ScaleFactor ≤ Vc(abs_asi))∧
          (¬∃ (j : nat):
              j < i∧
                (posn(bugs(asi)(j)) × ScaleFactor ≤
                  Vc(abs_asi)))) ⇒
        i = Cc(abs_asi)

rho_Needle((v : Speed)) : ASINeedle = (#posn: = v / ScaleFactor#)

rho_BugSeq((s : Speeds)) : ASISpeedBugs =
```

```
  Λ (i : nat) : (#posn: = s(i) / ScaleFactor, extent: = BugExtent#)

rho_Salmon((v : Speed)) : ASISpeedBug = (#posn: = v / ScaleFactor, extent: = SalmonExtent#)

rho_Scale : ASIScale = (#interpret: = Λ (a : Angle) : ScaleFactor × a#)

ρ((a : AbstractASI)) : ASIInstrument =
  (#needle: = rho_Needle(Vc(a)),
     bugs: = rho_BugSeq(Smm(a)),
     salmonbug: = rho_Salmon(Vref(a)),
     scale: = rho_Scale#)

in_arc((needle : ASINeedle), (astart, aend : Angle)) : bool = astart ≤ posn(needle) ∧ posn(needle) ≤ aend

next_counterclockwise : [[ASINeedle, ASISpeedBugs] → ASISpeedBug]

next_counterclockwise : AXIOM
    ∀ (needle : ASINeedle, bugs : ASISpeedBugs, bug : ASISpeedBug):
      (∃ (i : nat) : posn(bugs(i)) ≤ posn(needle)) ⇒
        (next_counterclockwise(needle, bugs) = bug ⇔
          (∃ (i : nat):
            bugs(i) = bug∧
              posn(bugs(i)) ≤ posn(needle)∧
                (∀ (j : nat):
                  j < i ⇒ posn(bugs(j)) > posn(needle))))

getCurrentBug((needle : ASINeedle), (bugs : ASISpeedBugs)):
  ASISpeedBug = next_counterclockwise(needle, bugs)

salmonBugCheck((needle : ASINeedle), (bug : ASISpeedBug)) : bool =
  in_arc(needle, posn(bug) - extent(bug), posn(bug) + extent(bug))

configBugCheck((needle : ASINeedle), (bug : ASISpeedBug)) : bool =
  in_arc(needle, posn(bug), posn(bug) + Smargin / ScaleFactor)

asiConfigCheck((asi : ASIInstrument)) : bool =
  configBugCheck(needle(asi), getCurrentBug(needle(asi), bugs(asi)))

END perceptualASI
```