Automating reasoning support for design

AUTOMATING REASONING SUPPORT FOR DESIGN

HESKETH, J.; ROBERTSON, D.; FUCHS, N.; BUNDY, A.

DAI Research Paper No. 823
Jul 1996

Submitted to Automated Software Engineering

Automating Reasoning Support for Design

Jane Hesketh, Dave Robertson, Norbert Fuchs, Alan Bundy

Abstract

Formalised design supported by automated reasoning can assist in the management of requirements - a particular problem for large, detailed systems. Designers developing an initial requirements into more detail and then producing a system specification must show not only that all the requirements have been met but also demonstrate how that has been achieved. This is especially important in safety-critical systems where sections of the requirements will be regulations or guidelines. Using real life examples from emergency shutdown systems for drilling rigs[1], we show how lightweight (and therefore less time-consuming) formalisation supports validation in an engineering approach to requirements management.

We have developed a requirements assistant - an interactive system for formalising and managing information about requirements including guideline requirements. As a design proceeds, relevant requirements are found automatically and checked before being notified to the designer with an accompanying explanation of whether or not they are currently satisfied. Progress in satisfying requirements is monitored automatically and contributing choices are recorded. Such evidence of adherence to guidelines is an assurance of the validity of the design. During any subsequent system modification, reference to this evidence can aid designers by drawing attention to the implications changes will have on maintaining guideline satisfaction.

This paper describes how this automated reasoning support works using the demonstrator we have built.

Keywords: Automated Reasoning, Requirements, Design, Safety-Critical Systems

1 Introduction

We set out to devise tools in the form of a requirements assistant (RA) which would assist designers and safety engineers in managing the process of working with large complex collections of requirements relating to safety critical systems. Such bodies of requirements are typical of systems which must satisfy regulations and standards. A priority was to find ways of supporting existing practise rather than to invent a new design methodology. However, practise does not stand still, as emerging standards are making new demands. In particular, designers of safety critical systems are increasingly expected to use formal methods to increase confidence in the validity of system specifications. Formal languages, although they place an extra demand on the designer, support the addition of semi-automated tools using techniques from Automated Reasoning and Knowledge Based Systems.

As our focus was on using formal language to manage requirements rather than express a specification, we dispensed with highly detailed precise logics, and adopted a more *lightweight* description. By lightweight, we mean a logic that has a rigorous formal basis, but only incorporates simple domain

[1] We gratefully acknowledge the collaboration and support of Shell Expro. This work was funded under the DTI/SERC Safety Critical Systems Programme on grant GR/J58619.

modelling. The user-defined part of the language, such as predicates and atoms may well be extensive, and responsibility for the semantics of this part of the language lies with the design team. We claim that considerable assurance can be gained using a limited logic which is far less time-consuming to learn and utilise than a full formalisation. By making the formalisation more accessible than heavyweight formal languages, it is likely that it will be used and understood more readily. Further, having a shared formalisation language serves to bridge the gap between engineering problem description and computer solution as described in [Robertson 96].

Responsibility for devising and meeting requirements is shared between the safety engineer, the designer and the RA. The RA provides tools to help safety engineers formalise guidelines for subsequent use by designers. During design, the RA monitors relevant requirements and assesses their satisfaction using proofs as arguments. These arguments are evidence of requirements satisfaction, but they lack the full force of proof in a detailed formalisation since the formalisation is lightweight and therefore has limited semantic content. Since the majority of designs failing to meet Shell's safety guidelines do so by failing to observe some obscure paragraph, simply noting the requirement's relevance is useful. The designer and safety engineer take responsibility for the semantics, ensuring that the lightweight formalisation is being interpreted appropriately for each aspect of the design. The RA provides a design tool for building a design with constant reference to the requirements. An instance is created of each relevant requirement as it applies to the design while under development. Each of these instances then needs an argument that it is satisfied, based on the design. For the finite world of an individual design many simple, often propositional, arguments are created as the design progresses, rather than complex proofs of general properties. The arguments record decisions, the factors which influenced them, and the contribution of components to requirements' satisfaction.

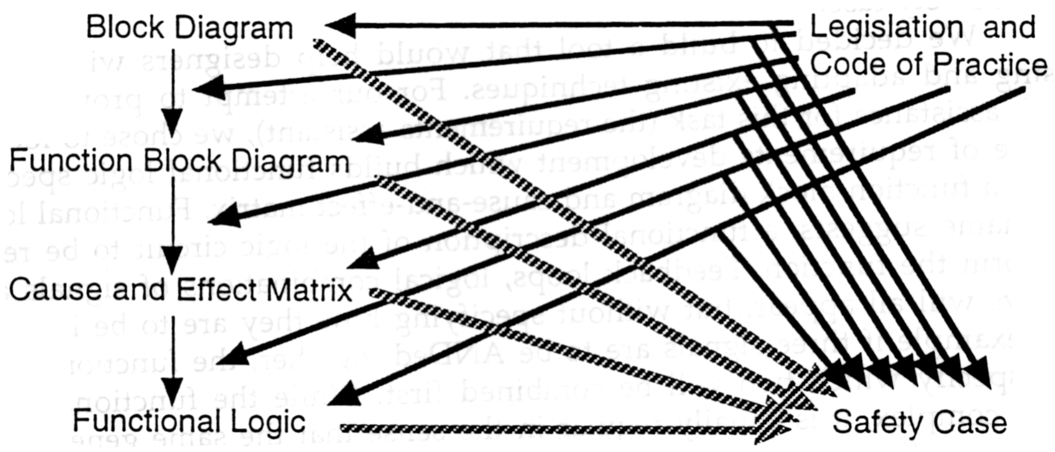
We used an example from the oil industry, that of designing emergency shutdown systems for drilling rigs. This meant that the system we built was engineered to suit that task, and tested against real problems. We did not aim for a general solution in theoretical terms, but by limiting our assumptions and making extensive use of modules and abstract data typing, we can make claims for generality of the approach. An advantage of working with a real design task is that there are standard components whose use is encouraged. Such components contribute known properties to the design which can be compiled in advance and used in the arguments for requirements' satisfaction.

Our demonstrator system is written in Prolog. We have encoded two central sections of "Fire And Gas Detection and Alarm Systems for Offshore Installations" (Shell 93) and some of "Emergency Shutdown and Process Trip Systems" (Shell 92). The example in this paper is based on the latter. The appendix illustrates formalised requirements from different sections of both codes of practice.

2 The Problem

2.1 Overall Task

Shell use four increasingly detailed levels of design to develop a specification of an emergency shutdown system on a drilling rig. The top level requirements statement is a block diagram which only identifies the major modules and the control flows between them. From this, designers must produce a far more detailed functional block diagram, describing instruments and equipment within the modules, and their control relationships to each other based on the states they can be in. The third level, the cause-and-effect matrix



details physical indicators (such as buttons pushed or high level sensor readings) of level two states, and the actions (e.g. equipment tripped) which must ensue when combinations of these indicating conditions occur. Finally, the bottom level specifies the functional logic to produce output signals which will effect the actions given input signals from the indicators/causes cited in the matrix.

However, the four levels of design description are only part of the picture. Designers must also satisfy the requirements covering all emergency shutdown systems which are set out in legislation, by the Health and Safety Executive (e.g. HSE 93) and in Shell's own codes of practice (e.g. Shell 92, Shell 93) which are commonly extensive and sometimes detailed. Ensuring the satisfaction of these general requirements is the responsibility of a safety engineer. They necessarily affect the choices made at all stages of the design, so designers must constantly be aware of them. For the eventual system to be certified by the regulatory authorities, a safety case must be provided showing why the system is believed to be safe within accepted limits. Naturally, the arguments will be closely related to the safety requirements.

When developing the design at each level in accordance with the earlier levels, Shell's designers must:

1. Notice when requirements apply,
2. Keep track of all the different requirements,
3. Assist with their interpretation in the current context,
4. Assess the extent to which they are currently satisfied,
5. Propose design decisions to address unsatisfied requirements,
6. Be able to provide evidence of requirements' satisfaction,
7. Be able to account for the roles played by different components,
8. Remember requirements' influence on the design decisions.

There are several reasons for remembering why decisions were made as well as what they were. Recording reasons at the time as opposed to inferring them afterwards makes validation clearer. As later design decisions should not undo previous work, designers must be able to be explicit about just what was being achieved, not simply how it was done. Modifications at a later date should not lose sight of requirements which had previously been satisfied by the bit to be altered and inadvertently cancel them. An account of all the design features which incorporate the safety requirements in codes and regulations contributes to the safety case.

We decided to build a tool that would help designers with these tasks, using and adapting existing techniques. For our attempt to provide designers with assistance for this task (the requirements assistant), we chose to focus on the phase of requirements development which builds functional logic specifications from a function block diagram and cause-and-effect matrix. Functional logic is, as the name suggests, a functional description of the logic circuit to be realised to perform the function. Feedback loops, logical combinations of signals and time delays will all appear, but without specifying how they are to be implemented. For example if three signals are to be ANDed together, the functional logic will not specify which two will be combined first. While the functionality may be quite complex, it is usually regular in the sense that the same general functions will be performed in many different circuits (e.g. all the startup bypasses) with only minor variation. Engineering practice supports such regularity through standardisation. For standard circuits, analysis and experience will already have established their properties. We can exploit this, reducing the amount of run-time inference needed. By treating whole functions as compound objects we can avoid considering their internal implementation details, and assume that these and their properties are known.

Conscious that designers would normally use computer tools to represent the various parts of their work in diagrams or text, we aimed to build a reasoning tool which could shadow the design process, supporting tasks 1-8 above. The RA would have to be able to produce functional logic under the designer's control, for inputs and outputs specified by the designer. It would have to advise the designer of the relevant requirements for the components under consideration, suggest ways to satisfy these requirements (e.g. circuitry), and construct arguments explaining whether or not each requirement was satisfied, with reasons.

Some preparatory steps were necessary. Reasoning would only be possible if requirements and systems descriptions were formalised. Both safety engineers formalising general requirements and designers would need suitable tools - to record their formalisations, and to help them reflect on the formalisations' meanings in the context of the original natural language: Shell's Code of Practice. Interfaces with the formal descriptions via the RA are needed. Housekeeping support for requirements management is also essential.

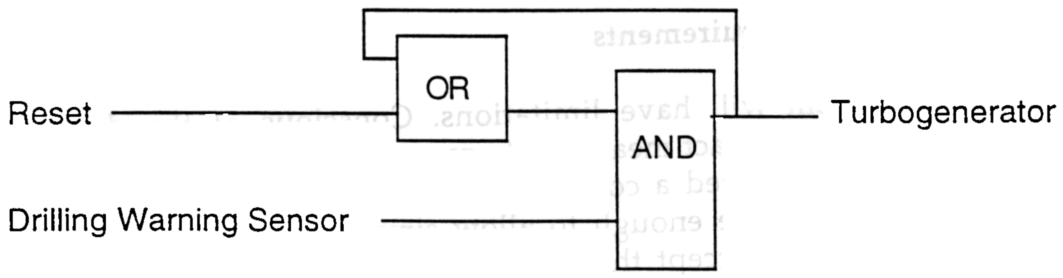
2.2 An example

The following example will be used throughout the paper, and should help make clear what we have done. We will look at the case of applying emergency shutdown control to a device when an error is signalled by a sensor. We have chosen a turbogenerator as the device, and a drilling warning sensor as the sensor. For this example we are assuming that the context is the evolving design of the emergency shutdown system, for which the designer knows of a cause-and-effect matrix, either from an earlier phase or under development.

Some general requirements from Shell's code of practice for emergency shutdown systems apply:

> *All trip demands cause shutdown of the appropriate outputs.* (G1)
>
> *On shutdown, outputs shall latch in the de-energised state until the initiating condition has cleared and a manual reset of the logic has been performed.* (G2)
>
> *Field devices may be reset. Reset shall be achieved by the provision of local push-buttons incorporated into each output control device.* (G3)

From all this information, a human designer would infer that the signal from any sensors such as the drilling warning sensor which should be able to trip a device (the turbogenerator) should be at logic level 0 when an error occurs and at 1 during normal operation. This is a normal practice which ensures safe operation in the event of a power failure. Standard logic to satisfy these requirements would have all the signals which could trip the same device logically ANDed, so that any single error could initiate a trip. To latch the error, the output of the AND gate would be fed back as one of its inputs, so that the error condition persisted. A reset would then be provided to override the latch once the problem had been fixed, and all the sensor signals had returned to logic value 1. In order to achieve the override, the reset value would also have to be 1 (when applied), and to cancel the 0 persisting from the error state, the reset and the latch should be ORed. Diagrammatically:



Although in this example only a single input sensor signal appears as an input to the AND gate here, there would usually be more.

Producing a specification like that above is only part of the designer's task. As discussed in 2.1, we need to offer the designer help to chose it and explain why it satisfied the requirements. A considerable amount of work is involved in typical design which will contain well over a thousand input and output signals. These may be combined in various ways, requiring different kinds of reset, startup, operator information and control, and maintenance override facilities, as determined by the Code of Practice.

3 Formalisation and Support Tools

3.1 Purposes of Formalisation

Employing formal languages makes it possible to provide the support facilities outlined in 2.1 because they enable automated reasoning and accurate information accessing. There are other compelling reasons for using formal languages. Formal languages are now commonly mandated by regulations. A significant contribution of formal methods is that the stage of expressing the specification in a formal language forces people to be explicit. The very process of doing this can make them appreciate aspects of the specification and their implications which had not been apparent before (Pfleeger 95). In our case, the functional logic diagrams are equivalent to a formal language, and the cause-and-effect matrix is an abstraction of them, but the Code of Practice is not normally formalised, and neither are the explanation of the choice of functional logic. As well as expressiveness, other advantages of using formal languages are their well-defined semantics, standardisation of language, and consequent easing of verification and validation. Disadvantages are their complexity, the copious work involved in applying them to produce a detailed description and lack of acceptance.

The unanswered question is: *which formal language?* To answer it, we must examine our intentions. Our goal in using formal methods might be termed *requirements management*. As specification must describe systems in full without ambiguity, languages capable of detail are normally used. Requirements management needs to access and reason with relevant information, some of it unprovable, normally distributed over many documents, which in turn may contain much that may be irrelevant to the current system. It must also offer traceability. In order to provide requirements management support, we only need a lightweight description of the problem, and hence lightweight formal methods. The designer does not have to work with the full detail of a precisely formalised design.

3.2 Formalisation of Requirements

Any formalism will have limitations. Conscious of the difficulties of conveying precisely the exact meaning of any natural language requirement in a formalism, we have adopted a compromise. We aim to work with a restricted language that will capture enough to allow simple reasoning and information access and control, and accept that it will only represent the design partially. Interpretation is the safety engineer's and designer's responsibility as it already is for current specifications and guidelines in natural language.

Our criteria for formalisation were as follows:

- We should use logic. Logics are expressive well-understood languages whose semantics are defined - which is essential if they are to expose misconceptions.
- Even though we could select some standard development language, such as Z, we should not anticipate which language will be used to express requirements.

Consequently, we modularised logic use, making different logics admissible in principle (see 7.6). Proof is managed by two modules in the RA. One lists the proof rules for the logic, and is only accessed by the other which is the automated prover controlling their use. Access to the prover is restricted to asking for proofs with their status (including alternative proofs). Our demands for constructiveness and decidability curtail the choices of logic considerably, though. However, beyond making some demands of the format of the logic (that it be sequent-based) and the kinds of formula which may be used in the hypotheses (see 4.1), there are no further constraints on the particular logic to be used. This is enough to allow us to deduce whether a requirement is relevant or satisfied in a given context. The demonstrator itself uses a restricted constructive first order predicate calculus with a sort hierarchy, to be explained in 7.5.

As noted above, it is expressly expected that safety engineers will take an active responsibility for their own interpretations. For example, suppose an engineer wishes to describe the control of a device (D) by a sensor (S). One requirement may be that on the sensor passing some threshold value, it should trip the device. A basic representation of this might be:

```
passed_error_threshold(S) causes trip(D)
```

There are obvious limitations in this representation. To name but a few: the error threshold is not given; the means by which the trip is effected are unspecified; whether the device actually becomes tripped or whether a trip is just initiated is ambiguous; and there is no mention of time. In spite of these deficiencies, the simple representation above is enough for basic requirements management provided that each predicate is well-defined and used consistently. It would certainly be enough to enable a network of causality relationships to be examined. It is also close to a level at which design requirements are presented, and while not offering a deep analysis, it offers a broadly comprehensive one. At present, the RA's support tools for terminology management are still primitive, and it relies largely on the same human discipline that engineers use now.

A fuller description of the requirement that a sensor detecting an error condition should trip some device might well fill in some of the gaps suggested above. To get to a precise description is demanding and the result complex, just in this simple example. Even then, however much the designer tries, the ensuing representation would still rely on their understanding and use of the defined language. This is an inescapable aspect of any description.

Our approach to description takes the designer's interpretative role between system and design as central, and puts her/him in the driving seat. Independently of formalisation or non-formalisation, the designer will interpret elements of the notation as standing for elements of the domain. There is inevitably always going to be a balance between the extent to which a safety engineer chooses to formalise, and the extent to which s/he relies on humanly declared and accepted but non-formalised definitions. Our intention is to leave this decision to the safety engineer, and to offer formalisation and reasoning support.

For the designer, using the formalisation is supported by:

- An interface which allows browsing through the satisfied and outstanding requirements;

- Links back from requirements expressed formally to the (viewable) natural language from which they originated, thus making it possible to see the formalised parts in context;
- A theorem prover which can provide complete and incomplete proofs, attempt automatic proofs;
- Display of the formal language either as sequents or in a simple translation to a pseudo-English translation;
- A tool for marking natural language text from requirements documents and linking them to their formal descriptions.

Thus it is open to whoever maintains the design support system to utilise a more detailed formalisation for critical parts of a system and less detailed ones elsewhere. Even for a single component, one aspect of its behaviour may be critical, while another may not. Response speed may be crucial when exact location is not, and the design language should be able to reflect this.

3.3 Support for Defining Requirements

The RA provides a tool allowing a safety engineer to read an English text online, mark phrases or sentences, and then record their translation in whatever formal language has been chosen. This process links each formalised requirement to a pointer to its original natural language text for subsequent reference. Requirement formalisation is a separate task from design. Once prepared, the file of formalised requirements can be loaded for application to a design.

3.4 Defining Requirements for the Emergency Shutdown Example

Shell's internal code of practice (Shell 92, Shell 93) was a major source of requirements in our task. It subsumes national regulations, by setting out many design practices to be adopted or avoided as well as goals to be aimed for, and is often quite explicit. It is written in English. The guidelines change infrequently after careful checking. A guideline formalisation would be stable, subject to inspection, and when altered could contribute to an understanding of the changed interpretations.

While laborious, it was fairly straightforward for us to create lightweight formalisations of natural language text. Shell's guidelines are written in clear simple language using regular well-understood terms, which lent themselves to formalisation using equivalent terms to those used in the text. Formalisation was used to write the existing guidelines in a regular form, not to expound them in more detail. Potentially, this is a weak point in the approach. Coming up with a suitable description language which models the domain simply is not easy. The sort hierarchy we used helped by encouraging the use of standard classes of objects. Standard operations such as tripping, shutting down and isolation were apparent. Regularity is crucial to an effective lightweight description because it enables the consistent use of a terminology shared by all. Otherwise, there is a danger that the limited terms include cases which are significantly different. Alternatively a confusing plethora of descriptions may be introduced covering subtle complexities without any structure available for argument - the disadvantages of heavyweight language without the advantages. Either of these problems is a likely source of miscommunication. In our examples, implicit knowledge sometimes had to be made explicit, but we would argue that that is useful if it exposes assumptions. The level of description used was consistent with our goal of fitting into existing work practices.

In the language chosen for our demonstrator, G1, G2 and G3 translate into four formalised requirements, R1 to R4, which are presented below with their English translations. To denote causation, an infix connective ->> is used, and ¯- links the description of the context in which this requirement holds (on the left) from the requirement itself (on the right). setof, which appears in R2, is a quantifier generating a finite set.

```
machine(X), initiator(I), expected_consequence(I,X)
⊢ tripped(I) ->> shut_down(X)
```

Context:	Given a machine X, and an input device I, which creates an input signal which is intended to control X,
Requirement:	an error signal from I causes X to receive a shut down signal.

```
machine(X),  setof(B,(initiator(B) & expected_consequence(B,X)),Bs)
⊢ reset(X) & (∀s  (on(s,Bs) & ok_signal(s))) ->> ok_signal(X)
```

Context:	Given a machine X, and the set of input devices Bs, creators of all the input signals which are intended to control X,
Requirement:	X being reset and ok-to-run signals being sent from all the elements which are Bs cause X to get an ok-to-run signal.

```
machine(X)
⊢∃ r local_reset(r) & resets(r, X)
```

Context:	Given a machine X,
Requirement:	there must be a local reset control which can reset X.

```
machine(X)
⊢ shut_down(X) & not_reset(X) ->> latched(X)
```

Context:	Given a machine X,
Requirement:	X getting a shut down signal and no reset causes X to be latched.

The requirements have been formalised in logic so that we can attempt to show each one to be true in each context in which it applies. The resulting set of proofs, when complete, will be evidence of the requirement's satisfaction wherever it applied. Each incomplete proof will indicate an unsatisfied requirement, and the gaps in the proof the nature of what is needed. The sequent style has been adapted giving hypothesis lists a special role - they express the context which must exist in order for the requirement to apply. The requirement will be considered to apply whenever all its hypotheses are satisfied, i.e. its context matches the current design.

These really are lightweight formalisations. They express a version of what is in the natural language version, but more regularly in logic, using a limited number of predicates. For example, recall that *G1* was: *All trip demands cause shutdown of the appropriate outputs.* As trip demands come from things which can initiate them (sensors, push buttons, etc.) and have effects on machines controlled by output signals, *G1* is translated as a requirement which applies when we have an initiator (initiator(I)), and a machine, (machine(X)), which is intended to be controlled by that initiator, (expected_consequence(I,X)). Whenever that is the case, the requirement is that if the initiator signals an error, then the machine should be shut down (tripped(I) ->> shut_down(X)). All the variables which occur in hypotheses (upper case letters) are universally quantified across the whole sequent.

We are making no claim that the formal translation we have chosen is a precise expression of the corresponding natural language. What is important is that the designer is alerted to anything in the Code of Practice which is relevant in order to make an informed decision. The extent to which safety engineers choose to formalise the Code of Practice and define the design language is at their own discretion. The 'proofs' of Code of Practice requirements are arguments and partial evidence, rather than having the force of proof they would have if a precise formalisation had been attempted.

3.5 Format of Formal Requirements and Use of Context

Having defined a set of requirements, a safety engineer will save them in a file for later use. Each formal requirement is recorded along with its origin in the natural language text (file and character position numbers). If necessary, there is a sequence number too, so that multiple formal requirements all originating from the same piece of text can be identified uniquely.

The formalised requirements are sequents, but their antecedents are not used in quite the usual way in the RA. Normally, antecedents are **assumed** to be true. We use antecedents to represent the context in which the requirement applies, in other words, we require the set of hypotheses to be true for the requirement shown in the sequent's consequent to be applicable. Therefore, the RA will **prove** that all the elements of a requirement's antecedent are true in order to establish that it is applicable.

Deciding what the context should be for a requirement is a balance between being specific enough and being too precise. If contexts are not precise enough, the designer will be deluged with irrelevant requirements. At the other extreme, it is no good being so specific that anything which did match the context already satisfied the requirement, for then no useful warning could be given. Consequently it is important that the requirement's context is minimal for describing the situation in which the requirement applies. Shell's Code of Practice is amenable to translation into straightforward requirements. Given a sentence from the code of practice, the basic approach is to write down the proof obligation as the consequent, and put any contextual information from the sentence into the hypotheses, along with sort information for any otherwise free variables.

To illustrate this point, consider requirement R3:

machine(X)
$\vdash \exists$ r local_reset(r) & resets(r, X)

which expresses G3: *Field devices may be reset. Reset shall be achieved by the provision of local push-buttons incorporated into each output control device.* Formulated as above, it is invoked as soon as there is a device X (machine(X)), and reminds the designer that a local reset will be needed ($\exists$ r local_reset(r)) which resets X (resets(r, X)). Suppose requirement R3 above had been formalised differently so that a local reset control for X figured in the context:

machine(X), local_reset(R,X) $\vdash$ resets(R, X)

R3^{o} says that if there is a device X (machine(X)), and an associated local reset for X (local_reset(R,X)) which does reset X (resets(R, X)). The two formulations are clearly not equivalent, but they are both possible partial translations of G3. R3^{o} would never be invoked until it was close to being true - because the reset control was already present in the design - and needed only be connected to the turbogenerator appropriately. By contrast, R3 reminds the designer both that a reset control must be provided and that it must have the desired functionality. It is also closer to the intent of the code of practice, since the existence of a reset is not normally optional.

Satisfying requirements will often have a domino effect, as adding more components to justify the argument for one may complete the context for another which can demand yet more components.

3.6 Design Components and Context

The set of components in the design is part of the definition of the world over which arguments about the satisfaction of requirements take place. The presence of a component in the design is in itself a fact in that world. Other objects contribute more facts such as functional abilities, e.g. circuitry. The RA needs these extra facts to try to establish requirements' satisfaction. To make them available, objects which can be added to a design are stored as schemata in files which are loaded into the system. Each schema has a name, a (possibly empty) context list, a list of components which constitute the object, and the factual contributions it makes to the design. A schema's context is used to instantiate variables which exist elsewhere in the schema, thus grounding the instance that is created and its contributions. The contributions are an important aspect of the system, because they encapsulate the properties known to be significant explicitly and concisely, rather than leaving them to be proved from scratch each time the object schema is used. So, a schema which incorporates a latch will announce that as a contribution, rather than leaving that fact to be inferred from its logic. Objects' contexts and contributions are not only expressed in the same formal language as the requirements, but are designed to meet them. Just as for requirements, each single contribution can generate multiple contribution instances, corresponding to each way that its hypotheses can be true. When an object is added to the design, as well as recording its presence, the RA instantiates and records the contributions it makes according to its schema. These contributions are then available as facts for the automatic reasoning module.

An example schema is a cable from the sensor to the device controller:

```
name:          cable
context:       machine(O)
               initiator(I)
components:    cable(I,O)
contributions: [] ⊢ tripped(I) ->> shut_down(O)
               [] ⊢ ok_signal(I) ->> ok_signal(O)
```

This schema may be applied if the design contains a machine and an initiator. On application, O and I are instantiated respectively throughout the schema. Its contributions are that when the initiator signals a trip, the controlled machine will be shut down, and when the initiator signal is OK, so is the signal to the machine. When an instance of an object is added to a design, the conclusion part of each instantiated contribution sequent is recorded as a fact available for inference, tagged by its contributing object. Further instantiation may have taken place when establishing the satisfaction of each contribution's hypotheses (if any), just as is the case for requirements. The component added which effects these contributions is a cable from I to O.

A more interesting schema is the latchedreset circuitry (below) which introduces a latch and links it to the reset control. The context in which it can be applied consists of a machine, its set of initiators and its local reset control. The latchedreset schema demonstrates the possibility of parametrisation, as Is can be a list of any positive number of initiators. Two components effect this schema which is the one shown in the example at the end of section 2. One is a multi-input AND gate which combines the initiator signals together with the latching and reset control. The other is the OR gate which allows a reset to override a latched error value. The free variable, Or, is the output of the OR gate, and is needed to show how the two components connect.

```
name:          latchedreset
context:       machine(O)
               setof(X, (initiator(X) & expected_consequence(X,O)),Is)
               reset(R)
               reset_control(R,O)
components     and([Or | Is],O)
               or([R,O],Or)
contributions  [on(I,Is)] ⊢ tripped(I) ->> shut_down(O)
               [] ⊢ (shut_down(O) & not_reset(O) ->> latched(O))
               [] ⊢ reset(O) & forall(j) : (on(j,Is) & ok_signal(j))
                         ->> ok_signal(O)
               [] ⊢ resets(R,O)
```

The latchedreset schema has four contributions, which can create more than four contributed facts when instantiated if there are multiple ways of satisfying the hypotheses:

1 Each sensor signal which signals a trip will independently cause the controlled device to be shut down. If there is more

than one initiator on the Is list, the RA will make an instantiated copy of this contribution for each one.
2. The device being shut down causes it to remain latched in that state while it is not reset.
3. All the sensors signalling an OK state, and the device having been reset causes the device to return to operation.
4. The reset which matched the schema's context is capable of resetting the device.

Using schemata encourages the benefits of standardisation. Engineers and designers working on designs and equipment should recognise standard configurations. Schemata save the designer from having to formulate contributions every time an object is added to the design. Recording objects' properties is a well-established technique, although it is less common for them to be functional properties. While we might have chosen to prove those properties from the design each time, that seems pointless when using standard schemata. We can reasonably expect such schemata to have tried and tested properties, which we take as given.

3.7 Definitions

Definitions are represented as implications in the formal language. The inference system allows a term which matches the consequent of a definition to be replaced by the antecedent. In this way, it is possible to use convenient shorthands. The replacing definition is not executed, for example as a calculation, just inserted into the formula. Definitions may not include recursion in order that finiteness and therefore proof termination properties are preserved.

An example we will use later is the definition of a local reset. A local reset control is one that is a reset, and the type recorded with it is that of a machine (i.e. the one it is local to) rather than a operations control centre:

```
reset(Type,X) & machine(Type) ==> local_reset(X)
```

The ==> symbol is non-material implication. If the antecedent is true then the conclusion is true. The definition appears to the left and the defined term on the right.

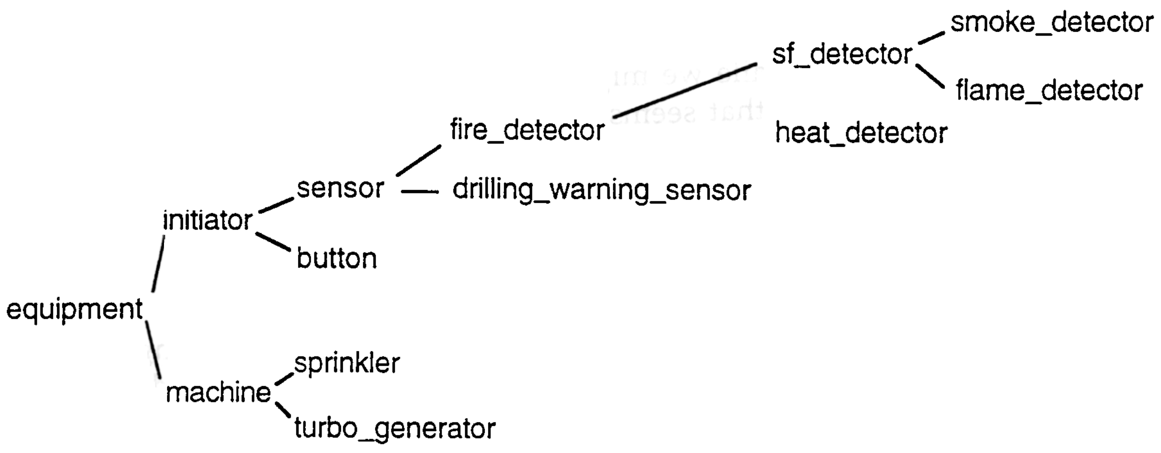
3.8 The Sort Lattice

Designers and safety engineers will routinely refer to classes of objects such as `initiators' or `devices'. In order to enable the same kinds of reasoning, the RA uses a sort lattice defined by the designer. Simple unordered types are not adequate, as many sorts break down into yet more sorts.

In our sort lattice, a node lying below another node bears a subset relationship to the higher node. Nodes are not parametrised. Since we expect the sort hierarchy to capture orthogonal kinds of sort information, it is a lattice, not a tree. Sort information about equipment may be whether it is pressure operated or electrically operated, and orthogonally may be whether or not it is part of the emergency shutdown system.

We chose not to build sorts into the demonstrator's logic as part of the syntax of quantification and hence the unification algorithm. Instead, we use predicates to describe them because we are aiming to use as general purpose a logic as possible. Although a little less efficient, we then achieve all the functionality of a sorted logic by explicit declarations in the context hypotheses which must true whenever a requirement or schema is applied. For example, the addition of a turbogenerator will result in the calculation of the set of all the sorts it is in. Using that set ({turbo_generator, machine, equipment}), the RA then takes as relevant those requirements which have a positively occurring predicate in their context which is an element of that set.

Here is some of the lattice used for the example later in the paper:



4. Requirements Management in the requirements assistant

In section 2, we gave several aspects of the requirements management task to be performed by the designer. Here we will explain how the RA functions to shadow these tasks and offer support.

4.1 Noticing when requirements apply

Usually each component which is inserted will have associated requirements. It is normally a designer's responsibility to know of any general behaviours and properties expected of the system and its components as well as their chief function. The RA is intended to help keep track of all such requirements as are formalised. Although we are dealing with general requirements such as Shell's Code of Practice here, specific requirements could be dealt with within the same framework. In our example, when the designer puts a device controller in the design, that should invoke reminders of general requirements such as G2 and G3: that the controller should be latched on error and that it should be possible to reset it.

The RA must be able to perform this task of finding relevant requirements too. Having located them, it should instantiate them in the context of the current design fragment. We establish relevance by making the formalised version of each requirement describe the context in which it applies, then the RA checks whether the requirement's context matches that currently under design.

Each requirement is formulated as a sequent consisting of a hypothesis list and a conclusion. The list of hypothesis formulae is used to represent the context in which the requirement applies, so if they simultaneously match a subset of the elements of the current design, the requirement as instantiated by the match is relevant. The hypotheses, usually literals, are implicitly conjoined, and all of them must be true at once in order for the context to match. Requirement R1

```
 machine(X), initiator(I), expected_consequence(I,X) |- tripped(I) ->>
shut_down(X)
```

is relevant when there are elements in the design which, when X and I are instantiated to them, make all the hypotheses true. The matching process may be a simple match against known facts, or may be proved. For example, the introduction of a turbogenerator called tg0, provides the fact turbo_generator(tg0), which from which machine(tg0) can be inferred, because the sort lattice states that turbo_generator is a subsort of machine.

The hypothesis formulae contain free variables (denoted by names beginning with capital letters, which are useful for pattern matching in the Prolog implementation) which also appear in the conclusion. The purpose of this use of free variables is to quantify over the sequent. As these variables are instantiated when matching the context to the design, their values are simultaneously set within the conclusion, setting up a (usually) propositional requirement formula to be established. The variables may range over objects in the developing design. No other quantification may appear in the hypotheses apart from set construction. The reason being that the role of the hypotheses is to describe a context in which the requirement applies, not to be a list of assumptions, so the hypotheses should only refer to actual objects which might exist and relationships between them. The free variables appearing in the hypothesis list are quantified. Each requirement's conclusion must be true for all instantiations of the free variables for which its hypotheses are true. For a requirement to apply there must exist values of the free variables such that the hypothesis formulae containing them are true. The conclusion is a single formula which may contain explicit universal and existential quantification over variables denoted by lower case letters.

The RA checks for relevant requirements incrementally, as each design step is taken. In order to find any new requirements invoked by the last addition operation, it reads sequentially through the potential requirements. For each, the RA establishes whether it matches the current context, by seeing if it is possible to prove all the hypotheses true at once as described above using the standard system theorem prover i.e. with a common unifier for the free variables. There may be no way of doing this, in which case the context in which the requirement applies is not present, and the requirement does not apply. There may be more than one way, in which case a different instance of the requirement will be created for each way in which the context can be instantiated and be true. Each is a distinct application of the requirement.

Returning to our running example, having placed a turbogenerator in the design, and nothing else, the RA works out that R3 applies. The instantiated requirement

$$\text{machine(tg0)}. \vdash \exists\, r : \text{local\_reset}(r)\ \&\ \text{resets}(r, \text{tg0}) \qquad \text{(R3')}$$

is added to the agenda of outstanding requirements. The proof which allowed the context to be established is not kept, although it could be in a more sophisticated system. Similarly, R4′ (from R4, and originally G2) is added as an outstanding requirement:

$$\text{machine(tg0)} \vdash \text{shut\_down(tg0)}\ \&\ \text{not\_reset(tg0)} \text{->>} \text{latched(tg0)}$$

Some requirements referring to design components will not be invoked until the context in which they apply is complete. The formalisation R1 of G1:

$$\text{machine(X), initiator(I), expected\_consequence(I,X)} \vdash \text{tripped(I)} \text{ ->> } \text{shut\_down(X)}$$

does not apply yet, because while machine(X) can be made true when X is tg0, there is not yet a value for the variable I which will make initiator(I) true.

The order of elements in the context is not significant. In case later context elements can provide information which might make it possible to establish earlier ones, the RA attempts to prove the set of them in all necessary permutations until it finds a proof or has exhausted the possibilities. By choosing to prove first those context elements which will not need to wait on others' satisfaction, this process need not be inefficient.

4.2 Keeping track of all the different requirements

In addition to letting designers load a list of applicable requirements at startup time, the RA maintains the lists of all the satisfied and unsatisfied instances of requirements being invoked.

Each time a requirement is applied, that instance of its application is recorded. If the RA establishes that the requirement is satisfied at that point, it is added to the list of satisfied requirements along with the argument for its satisfaction. If the RA concludes that the requirement is not satisfied, the instance and any partial argument relating to its satisfaction are put on a list of outstanding requirement obligations.

Sometimes the new design step will cause the status of requirements already on one of the lists to be revised. The new step may affect only the argument for a requirement's satisfaction, in which case the action is straightforward - the argument is reevaluated and rerecorded. If necessary the requirement (with the new argument) is moved to the other list. A more complicated situation arises if the context which caused the requirement to be invoked changes. Then the RA applies the requirement afresh as demanded by the current context and finds and cancels any superseded versions. This happens if, for example, the design has added to the world so that the context match which was formerly true is now only a subset of the current match. Take the requirement:

$$\begin{aligned}&\text{machine(X),}\ \ \text{setof(B,(initiator(B) \& expected\_consequence(B,X)),Bs)}\\ &\vdash \text{reset(X)}\ \&\ (\forall s\ \ (\text{on(s,Bs)}\ \&\ \text{ok\_signal(s)})) \text{ ->> } \text{ok\_signal(X)}\end{aligned}$$

which describes the property that a device controller which has been reset should have an OK-to-function signal when all its sensors are also signalling OK-to-function. When only one initiator is present, say a drilling warning sensor, dws0, Bs is just the set {dws0}. But if another initiator, a flame detector fld0, is added which is also capable of tripping tg0, the set of initiators for tg0, which was just {dws0} before, is now {dws0, fld0}. The old requirement referring to {dws0} is out-of-date. The RA must detect that the old instantiated requirement has been superseded.

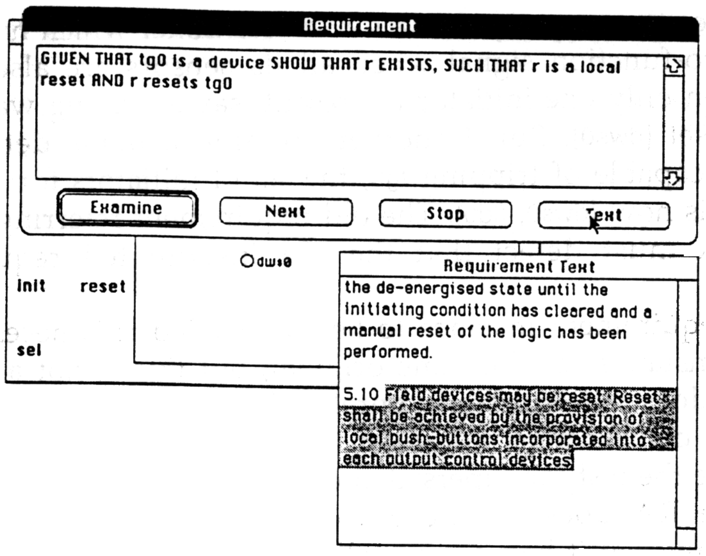
For a requirement instance to be superseded it is necessary that new and old are both instances of the same original, but this is not a sufficient test. The new instance does not supersede unrelated instances of the requirement, so the new instance's context must also refer to at least some of the same context as the old one. While the test for superseded context could be detailed, it need not be in our case. The RA does not need to know how the requirement has been superseded, only that it has, since replacement is easy to compute. The test for a context overlap is simplified because we are comparing two instances of a common context list. All corresponding pairs of context list elements should be alpha equivalent except for those which indicate subsumption, of which there must be at least one. The subsumption indicators are set generators producing intersecting sets in the new and old instances of the requirement.

From the point of view of a designer using the system, it may be laborious to see many requirements being superseded when apparently little change has taken place, but important, as they are all valid.

4.3 Assisting with requirements' interpretation in the current context

At any time, the designer may review the state of the requirements, whether shown or outstanding. Our expectation is that as well as reviewing its status, the designer should also be conscious of the interpretation of each requirement. To this end the RA shows a simple pseudo-English translation of the formal statement by default. Additionally, the designer can choose to inspect the instantiated sequent with its proof (see the next section), and the original code of practice from which it was derived. Our aim is to assist the designer in maintaining consistent, accurate interpretations and remaining sensitive to the intentions and goals of the original code of practice.

The next figure shows the designer viewing invoked requirement R3'. The designer has opted to check on the original text too, by clicking **Text**. The highlighted part is the original text from which the formalisation was obtained and it appears in the context of the original document.


GIVEN THAT tg0 is a device SHOW THAT r EXISTS, SUCH THAT r is a local reset AND r resets tg0

the de-energised state until the initiating condition has cleared and a manual reset of the logic has been performed.

5.10 Field devices may be reset. Resets shall be achieved by the provision of local push-buttons incorporated into each output control devices

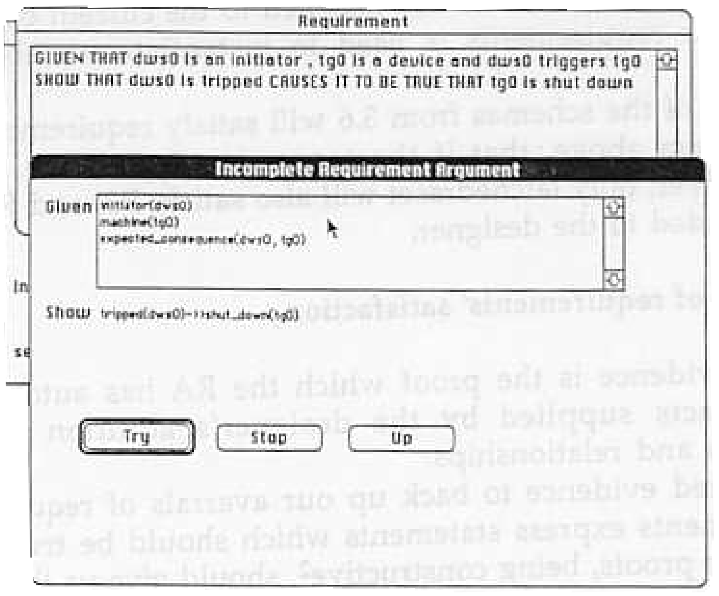
Translation into pseudo-English is accomplished by means of a Definite Clause Grammar (a standard language translation utility in Prolog, described in Clocksin 94), where logical connectives and predicates have language translations defined when building the requirements. Translation of the sequent structure itself is standard. The requirement sequent is presented as **GIVEN THAT** followed by a list of translated context elements, then **SHOW THAT**, and finally the translation of the sequent's conclusion - the requirement to be proved. The context literal machine(tg0) is translated as the argument **tg0** followed by the translation of the predicate: **is a device**.

Given a formalised requirement instantiated to the objects in a design context, the designer will carry the final responsibility for the requirement's interpretation and satisfaction. Since we are not expecting a total formalisation, that responsibility is shared between the computer system taking care of syntactic reasoning on formalised parts of the code of practice, and the human designer, who must monitor the semantics. In one way, the more detailed the formalisation, the less open-ended the human's task is, as each term in the language will be more precisely defined. On the other hand, a more elaborate description can be harder to follow, increasing the possibility of mistakes. If the formalisation is less comprehensive, the designer has to do more to interpret the requirement according to the domain and the instance.

Being able to refer back to the original textual goal is particularly important for helping designers monitor their formalised interpretations.

4.4 Assessing the extent to which requirements are currently satisfied

In the last figure, clicking on **Examine** would allow the designer to examine the requirement's formal proof, whether partial or complete. Below, R1' is shown in its formal logic, along with its proof, which is currently incomplete. The designer may examine and work on these proofs interactively. The box marked **Given** contains the sequent's hypothesis list, and its conclusion follows **Show**. The sequent shown is a leaf, i.e. it is on the fringe of the proof tree developed so far where it no longer breaks down into more detailed subproofs. If further proof were available below, a button would be presented allowing the designer to navigate down the proof tree. An example is in section 6.3.



Requirement

GIVEN THAT dws0 is an initiator , tg0 is a device and dws0 triggers tg0
SHOW THAT dws0 is tripped CAUSES IT TO BE TRUE THAT tg0 is shut down

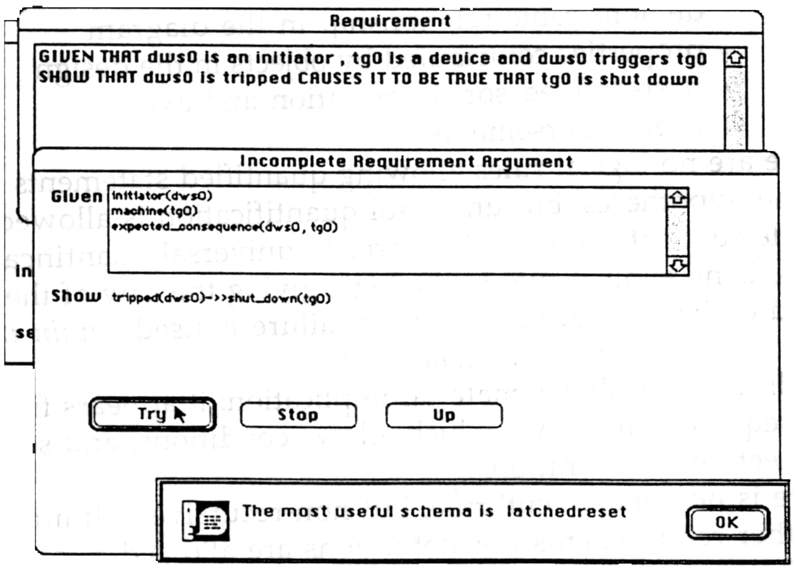
Incomplete Requirement Argument

Given
initiator(dws0)
machine(tg0)
expected_consequence(dws0, tg0)

Show tripped(dws0)->>shut_down(tg0)

4.5 Proposing remedial design for unsatisfied requirements

Since at any stage, the designer has the obligation of satisfying outstanding requirements but is only allowed to achieve this by inserting prescribed components from the RA's database, the RA can offer advice by second guessing. For a given outstanding requirement, the RA can use abduction to see if the addition of one of the known components has the potential to complete the proof. Each component schema records contributions made by its addition to the design over and above its presence, for example, connectivity. In the example, while looking at the details of R1′, the designer asks for such advice:



Requirement

GIVEN THAT dws0 is an initiator , tg0 is a device and dws0 triggers tg0
SHOW THAT dws0 is tripped CAUSES IT TO BE TRUE THAT tg0 is shut down

Incomplete Requirement Argument

Given
initiator(dws0)
machine(tg0)
expected_consequence(dws0, tg0)

Show tripped(dws0)->>shut_down(tg0)

The most useful schema is latchedreset

The advice is calculated by looking at the applicable schemas' contributions and seeing which schemas satisfy the greatest number of outstanding requirements in relation to the current context. Of these, we prefer solutions which do not add unexpected contributions, since that would be likely to introduce unnecessary complexity. In the current example, two schemas apply, i.e. their context formulae can be matched to the current context (this process, as with that for requirements is used to instantiate variables elsewhere in the schema).

Either of the schemas from 3.6 will satisfy requirement R1′ instantiated as in the diagram above, that if the sensor is tripped, the device becomes shut down. However, only latchedreset will also satisfy R3′ and R4′. For that reason, it is recommended to the designer.

4.6 Evidence of requirements′ satisfaction

The evidence is the proof which the RA has automatically constructed using the facts supplied by the designer's addition of objects and their contributions and relationships.

We need evidence to back up our averrals of requirements' satisfaction. The requirements express statements which should be true about the designed system. Their proofs, being constructive[2], should give us this evidence.

There is one respect in which our use of the term constructive here differs from the classical one: we accept rules of negation-as-failure[3] because we are working in a finite world. We chose a sequent calculus style, because it usefully distinguishes the context in which the conclusion of the theorem is to be true - a key feature of our approach. We need to be able to decide when a requirement is applicable, and exactly what part of the design is the context it is to be applied to.

The restrictions used in the demonstrator to achieve a logic with the desired termination properties are:

- only a single conclusion is allowed;
- proof leaves are justified by any of
    - ♦ facts in the context - objects which are in a diagram or in some demonstrable relationship in the diagram
    - ♦ properties contributed by objects in the design
    - ♦ user-defined sort information and axioms
    - ♦ user endorsements
- there are no logical rules allowing quantified statements to be unpacked in the hypotheses, but universal quantification is allowed across the whole sequent - effectively a prenex universal quantification;
- there is no explicit rule in the logic giving the 'law of the excluded middle' directly, but negation as failure is used *within the finite context with which the proof is concerned;*
- implication disallows material implication: it appears firstly by virtue of the sequent connective, which allows conditions, and secondly as the connective in definitions;
- there is no general 'cut' rule, but non-recursive definitions are allowed;
- Neither requirements nor definitions are allowed to contain any implications which contain quantified statements in their contexts or antecedents. Set generation is allowed, because it generates a finite set

[2] A constructive proof is one which only infers from things which exist or can be constructed from things which exist. Constructive proofs do not admit the law of the excluded middle, for example.
[3] A negated formula is deemed to be true if the formula cannot be proved true - see Clocksin 94.

from the finite world.

These restrictions are similar to the standard ones used in a sequent calculus to turn full FOPC into a constructive logic. The first difference is the further restriction to disallow quantification within the hypotheses. To allow us to assume another universal or existential truth, without proving it would not be advisable in a safety-critical setting. The second is to choose a version of negation rules which differs from the constructive ones which we claim is justified by the proviso that this is permissible in finite worlds.

The use of sets allows us to define a context within a context. setof is a defined connective. It is effectively a local universal quantification, whose scope in the finite world can be specified.

These restrictions have not proved problematic within our examples, in that we have been able to formalise any requirements we have attempted, therefore we believe that they are not unduly restrictive.

4.7 Accounting for the roles played by different components

Designers need to know the roles played by components in satisfying requirements in order to know what the implications are of changing or removing them. The constructive proof for each requirement records the contributions made by all the components involved. The contribution of any component can be inferred by looking at all the requirement arguments in which it occurs.

4.8 Remembering requirements' influence on the design decisions

As well as knowing components' contributions to meeting requirements, designers need to know which requirements have been incorporated and how they have been satisfied by component choices. The influence of each requirement is apparent from all the instances of it which appear as requirement justification arguments. All these arguments are dependent on the components that appear in them.

5. Using the requirements assistant

On starting the RA, the designer selects files of requirements, components and their sort hierarchy. Then the designer must create a design and define the world for which requirements will be monitored via the RA, so that it can register everything. We expect design definition to happen through RA-controlled interfaces which only allow the designer to use known components. Such interfaces must be provided for designers using the RA, corresponding to different viewpoints. For our demonstrator, we developed two simple interfaces, one for physical layout and one for functional logic.

The demonstrator RA lets a designer describe an emergency shutdown system topologically, using a window called the **Diagram Editor**. The RA has a tool menu which includes the description of other objects as well as the ones we have used in this paper: doors, rooms, control panels, sockets, routes etc. For defining circuit connections, the designer uses the **Functional Logic** window. In last of its set of context formulae is now expected_consequence(dws0,tg0). Although the RA will try to deduce the truth value of expected_consequence(dws0,tg0) from the facts known from the diagram, it cannot. Whether or not expected_consequence(dws0,tg0) is true depends on the designer's intention, as only some of all the initiators present in a design will be expected to be capable of tripping a particular device. As information which can only come from the designer, we expect that expected_consequence(dws0,tg0) will either have been declared to be true e.g. given as part of the cause-and-effect matrix, or be something the RA can ask of the designer.

The RA will see if there is an axiom, which might, for example, have been declared as part of a cause-and-effect matrix. Failing that, the RA can see if the predicate is declared as "askable". expected_consequence is defined as askable. Askable predicates are protected by the condition that their arguments must be fully instantiated before the designer is asked whether or not that instance is true. expected_consequence(dws0,tg0) is ground, so the designer will be prompted as to whether it is true or not.

The text for the query is automatically generated using the same natural language generator used for presenting requirements. If the literal expected_consequence(dws0,tg0) is asserted to be true by the user, that fact is added to the world's facts along with its origin - from the user. If the designer were to select "no", the equivalent falsity would be recorded. The reason for this is that now the fact is known, we do not wish to ask the user again. In our example the designer selects the answer "yes", and the RA attempts to prove

machine(tg0), initiator(dws0), expected_consequence(dws0,tg0)
$\vdash$ tripped(dws0) ->> shut_down(tg0)

but still fails, because it cannot prove the truth of the conclusion. R1' is added to the agenda of outstanding requirements. Although we will not do so here, we could add more initiators and get more copies of this requirement, instantiated appropriately.

The other outstanding requirements are now:

machine(tg0), setof(B,(initiator(B) & expected_consequence(B,tg0)),Bs)
$\vdash$ reset(tg0) & ($\forall$s (on(s,Bs) & ok_signal(s))) ->> ok_signal(tg0) (R2')

machine(tg0) $\vdash \exists$ r local_reset(r) & resets(r,tg0)

machine(tg0) $\vdash$ shut_down(tg0) & not_reset(tg0) ->> latched(tg0)

While the contexts might appear redundant at this stage, we keep them in their role of hypotheses, bearing in mind possible future developments of the RA. Many logics use hypotheses more actively than the demonstrator's current one, and we want to keep that wider use an option. Further, one way of speculating about the effects of future design steps is to add their effects to the hypothesis list temporarily and check the result. We wish to preserve that option too.

The reset is added similarly to the first two components. Resets are of two types in the demonstrator. Central ones operate from the main control centre

(**ICC**). Local ones are for a given device. When designer selects **reset** from the tool palette, the set of devices on the current context (the current Diagram Editor window), is automatically constructed. The union of that set and **ICC** is displayed in a menu for the designer to choose from. When the reset is inserted, the designer's menu choice will be recorded with it. From this information, definitions allow the RA to deduce whether a given reset is local because it is devoted to a particular device, or whether it is a centrally controlled reset which can affect many devices at once. Such choices are significant for proofs of requirements demanding different kinds of reset control. The reset's name (**R-tg0**) is generated incorporating that information, too, for the convenience of the designer.

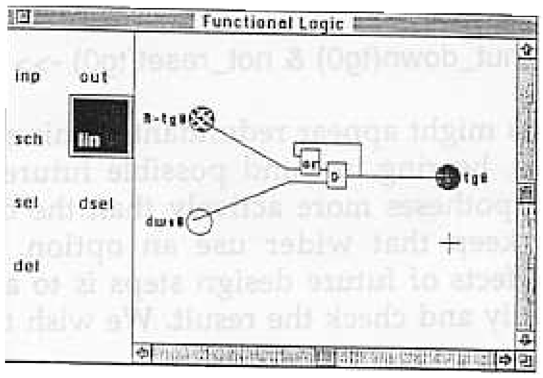
Lacking the connecting circuitry, the RA cannot complete proofs at this stage. On request however, it can advise the designer by trying to look one step ahead and see if any known component would be useful, as described in 4.6.

6.2 Inserting Components using the Functional Logic Editor

The designer switches to a functional logic editor by a menu bar selection. The RA displays a new window with a logic tool panel which allows the insertion of inputs, outputs and schemas, again prompted by menus of alternatives.

Although the functional logic view operates in the same context as the Design Window there is no assumption that all the components present there will be involved in this logic. The designer must specify them. However, when the designer selects **inp** from the tool panel, the RA computes that the possible inputs in the context from a functional logic point of view are any initiators and reset controls. In our example they are dws0 and R-tg0, so the RA offers them in a selection menu. Having inserted both of these input signals and tg0 as an output signal, it's time to connect them by a circuit schema.

Selecting **sch** from the tool palette results in a choice of schemas. Choosing a latchedreset from the menu (which also includes AND and OR gates) places a diagrammatic representation of one on the window (see below), adds a named instance to the world (schema0), prompts the designer to assign its connections (thus setting its context), adds all the schema's contributions to the domain, and as with the addition of any component, revises the lists of known and outstanding requirements.



These are the (instantiated) contributions added by choosing this schema:

```
tripped(dws0) ->> shut_down(tg0)                                     (C1)
shut_down(tg0) & not_reset(tg0) ->> latched(tg0)                     (C2)
reset(tg0) & forall(i)  (on(i,ls) & ok_signal(i)) ->> ok_signal(tg0) (C3)
resets('R-tg0',tg0)                                                  (C4)
```

It is worth noting that these contributions encapsulate several examples of input/output behaviour. Each of the ->> causations does, and so do the latched and resets predicates. By describing the contributions of an object or logic circuitry at a high level, details of its temporal properties or quite how it is achieved can be finessed to a large extent, while preserving essential behavioural information. This is a regular feature of the lightweight descriptions we have adopted. For example the RA's valve schema allows it to infer that the closure of a valve on the pipe between two vessels will isolate those vessels from each other. Thus the agents and effects are noted without modelling the process in complex detail.

6.3 Completing Requirement Proofs

In addition to completing the design, the designer has now satisfied all the requirements. The system's reassessment of the state of the requirements confirms this.

The proofs of all the requirements apart from R3' are trivial because their conclusions match contributions directly, modulo renaming of variables. In R2, Bs will have been calculated to be {dws0}. Their proofs will have leaves which record that the conclusion has been contributed as a fact by schema0.

Proving R3' takes a little more work. For ease of reference it is given again:

machine(tg0) $\vdash \exists$ r local_reset(r) & resets(r, tg0) (R3')

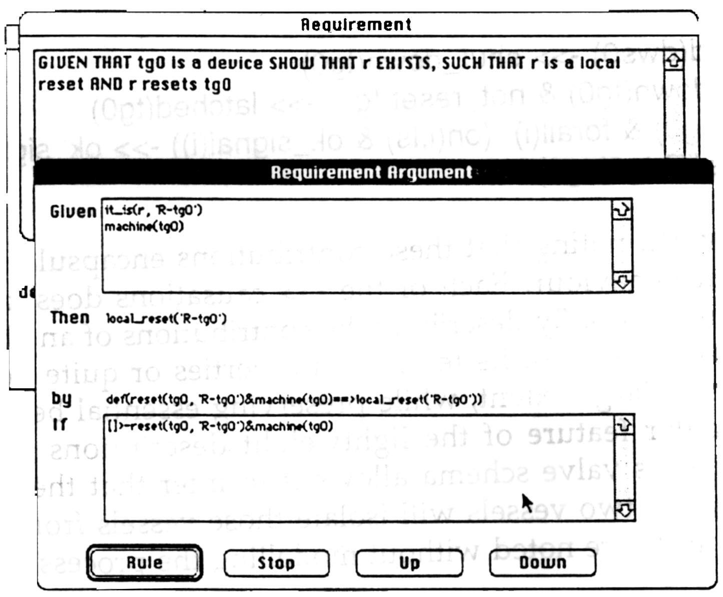
Usually in a sequent calculus the proof at this stage would demand that an explicit value be provided for r. To find a suitable value for r, the theorem prover makes it a Prolog variable (R, say) and proceeds to try and prove the existentially quantified formula, to see if the demands of the proof steps which follow will force a value for R, and therefore for r. That value is then used in the proof rule for $\exists$. This is in fact what happens here. The prover attempts to prove:

machine(tg0) $\vdash$ local_reset(R) & resets(R, tg0)

which is immediately split into two subgoals which must both be proved:

machine(tg0) $\vdash$ local_reset(R)
machine(tg0) $\vdash$ resets(R, tg0)

the first subgoal can be proved true by setting R to have the value R-tg0 (as we shall see next). With this value, the second subgoal is immediately true from contribution C4. The order of the subgoals is not significant. The prover will try to prove them in all permutations until it finds a solution or fails completely.



GIVEN THAT tg0 is a device SHOW THAT r EXISTS, SUCH THAT r is a local reset AND r resets tg0

```
Given  it_is(r, 'R-tg0')
       machine(tg0)
Then   local_reset('R-tg0')
by     def(reset(tg0, 'R-tg0')&machine(tg0)==>local_reset('R-tg0'))
If     []>-reset(tg0, 'R-tg0')&machine(tg0)
```

The prover uses the definition of local_reset which requires that R be a reset for something and that that something be a machine in the current context. The proof is again split into two subgoals, which are proved from the context. The overall proof of R3' has been produced by the RA from piecing together information from the designer's explicit additions to the design and by inference from contributions due to some of these choices.

7. Discussion

7.1 Finite World and Focus on Context

A common problem in automated reasoning is controlling search. Problems which can be described finitely have the significant advantage that proof attempts terminate. We have taken advantage of the concrete nature of our task to force the domain to be finite. Where recursion occurs in the system being described, it has been deliberately encapsulated in order to preserve finiteness within the description language. For example, in the latching circuit the property of providing a latch is given without going into the recursive detail of how this is achieved.

By building a notion of focus into the system, we can limit still further the size of the world to be considered for each task, whether it be checking the satisfaction of a requirement or establishing which ones are relevant. The notion of focus echoes the way a human designer would proceed - looking at one design module at a time, or one page of circuit diagram at a time.

7.2 Proof Strategies

By working in a finite domain, and restricting the use of recursion and the cut rule, we have an almost propositional system. Although requirements may be universally quantified, that quantification will only be over a finite domain, so we need only ever prove the finite set of instantiations of each requirement formula to all be true. Existential quantification can appear, but it, too, is constrained to a finite domain which would contain any candidate value.

The sort declarations are purely a set of subset links which are used transitively for deduction. Proof search using transitivity is prevented from becoming explosive by demanding that the intermediate sort be instantiated before we embark on the subproofs. In this way, with a finite lattice, sort proofs are terminating.

This finiteness allows us to use negation as failure safely, since we can apply proof exhaustively. So far we have worked with relatively small design and requirement sets and not encountered efficiency problems.

7.3 Using sorts to represent partial descriptions

The entire set of subsorts may include sorts which are not leaves of the lattice, they are still sorts of sorts. Choosing one of them corresponds to partially specifying a component. For example a sensor may be declared to be a heat sensor, but without deciding whether it is a flame detector, a temperature sensor or a smoke sensor. Our system does not yet monitor such partial descriptions, but they could be exploited in a system using a refinement logic.

7.4 An Evolving World

As the design proceeds the context is fleshed out, and the domain of proof changes. Adding components can change simple facts and add lemmas to the theory as components contribute properties. This means not only that proofs which were unprovable become provable, but that proofs which were true may need to be reproved. As we have seen, a proof that was true for all the sensors in an earlier partial description of the system will not suffice when a new sensor is added. As described earlier, the RA monitors this process and adjusts superseded requirements.

Our solution to managing an agenda of outstanding requirements lies between full truth maintenance and full regeneration. Full truth maintenance would involve comparing the proofs of each of the context elements in the two instances of the requirement. We do not need such a level of detail. For our purposes it is enough to know that the context[4] of the earlier requirement instance has been subsumed. This is so if the values for the set of context variables yielded by the current proofs subsume those of the earlier proof (or vice versa). If the context has changed in this way, the RA uses the new requirement instance as a replacement for the subsumed one. In this way regeneration has taken place, but only of a requirement detected in relation to the addition. The test for replacement need only establish context subsumption.

Full regeneration would have the advantage of avoiding the truth maintenance problem completely, but demand the regeneration of *all* the requirements (shown and outstanding) each time a component was added, not just those requirements pertinent to the addition. For a large system, this could prove expensive. Another disadvantage of full regeneration would be the failure to offer a designer information about the implications of an individual design

[4] Here we use 'context' to refer to the set of components which are the context for the requirement - the ones to which its variables are instantiated.

step. For this reason the implemented approach is closer to the human designer's reasoning.

7.5 The Demonstrator's Logic

For the purpose of developing an exemplar we chose a lowest common denominator logic, one that would be a subset of almost any other logic. We chose a restricted first order predicate calculus with sets. There is a causation operator, which is simply an infix predicate (->>).

While specialist non-classical logics have been developed, they are commonly more complex than ordinary first order predicate calculus. Choosing a vanilla logic allows us to show the value of our approach without relying on special purpose language features. Moreover, automated strategies for proof, which we *did* want to rely on, are well-developed for standard logics.

Rules relating to the physical interface use a logic of two dimensional relations allowing the designer to specify topological relationships. As well as detecting the presence of objects, the RA can detect *within, at* and *leads to/from* relationships.

Referring back to the process of monitoring relevance of requirements, it is worth observing what would happen if both material implication and negation-as-failure were in the logic. At the stage when only the turbogenerator had been added, but no initiating sensor, R1 would have one true and two false hypotheses using negation-as-failure:

machine(X), initiator(I), expected_consequence(I,X) $\vdash$ tripped(I) ->> shut_down(X)

Material implication would then permit the inference that the conclusion was true, which would be meaningless.

7.6 Engineering the Demonstrator

Wherever possible, we adopted an abstract data typing approach to building the system, in order to assist development and make the demonstrator as modular and easily extendible as possible.

The RA uses abstract data typing when accessing the proof system. It must be able to ask whether or not a sequent is proved true, and which objects are involved in the proof. An organisation using such a system which decided not to use the demonstrator's vanilla logic would be responsible for providing rules of inference for the chosen logic, and adapting the proof strategy module. It would specify predicates, connectives, axioms and constants for designers to use.

8 Evaluation and Further Work

As we said at the beginning of this paper, our intention in devising the RA was to provide a tool to "assist designers in managing the process of working with large complex collections of requirements relating to safety systems". We focussed on the core tasks, their representation and automated reasoning support. User interfaces were sufficient for demonstration purposes, but not a focus. As well as evaluating our attempt on the core tasks, we can now look beyond to wider questions of usability, in the light of experience with demonstrator.

8.1 Lightweight Formalisation

Our use of lightweight formalisation was a key component in achieving both validation arguments and automated support for managing them and their associated requirements. We must consider whether it was effective for expressing requirements.

For each of the different sections of the guidelines we chose, we developed a vocabulary. Some vocabulary stretched across different sections, referring to commonly occurring objects such as an alarm or the control centre. We wrote a set of simple formalisations of the natural language requirements using high level terms which had a close to human level interpretation in the domain. Linking formalisations to designs was largely straightforward.

We also need to know if lightweight formalisation itself scales up, and what effect scale has on vocabular. The description language can usefully be thought of in four parts: the sorts, properties of individual objects, the relations and the logic itself. The sorts are simple constructs describing individual objects. Properties, are similarly simple describing heights of buttons, and colour-codings for example. A simple logic was chosen deliberately, as part of the lightweight formalisation, in order to be readily comprehensible, and it was constant. Relationships (causal or locational, for example) are the most challenging part of the formalisation, because they require significant proof. To assess the viability of lightweight formalisation of a whole domain, we progressed from our initial formalisation of the design criteria for emergency shutdown systems from the Emergency Shutdown guidelines (Shell 92), and encoded the guideline requirements for a whole fire zone[5]. The second phase of formalisation built substantially on the first one, confirming our belief that lightweight formalisation was able to capture common concepts. Fresh terminology required for the new domain was mainly to cover sorts and properties. New relationships were rare.

Demonstrating the RA and typical formalisations to a domain expert was encouraging in terms of understandability. For usability, it would be important to ensure consistent use of language, especially when referring to events and relationships. Vocabulary management assistance would be invaluable.

8.2 Extensibility to a real-life setting

Industrial Usability

The RA would have to be integrated into the actual CAD system used by a

[5] sections on "Manual Alarms and Executive Actions" and "Fire Detection and Annunciation" in the Fire and Gas Detection and Alarm Systems guidelines (Shell 93)

system designer, as exemplified in the demonstrator. It is unlikely that designers would willingly add an extra layer of work to run their designs through an separate requirements management system, which would then provide them with relevant information too late.

The existing support tools for the safety engineer, for example for generating formalised requirements, are basic. Ideally there would be a structure editor built around the logic and vocabulary. Requirements definition and development should also notice interacting requirements. Requirements may be rightly duplicated, because different parts of the guidelines repeat them. More importantly, it would be useful to notice if they were contradictory. Such tools could also allow separate formalisations of new and old guidelines, or internal and external regulations, to be compared.

A better interface would be needed to assist in accessing the volume of requirements that are invoked even by simple designs. Modularisation would be essential. An instance of an accommodation unit with an exit leading onto the top of a stairway leads to the activation of 28 fire zone guideline requirements from one section of the guidelines alone in the current formalisation. This happens because satisfying the initial guidelines results in adding further features, which in turn trigger more requirements.

Full Scale Tasks

Scaling up can involve not only a larger design, but more requirements and the demands of a wider domain on the formalisation language. The latter issue has already been addressed above. Now we will look at the demands of more extensive designs.

Requirement checking has two phases: adding newly invoked requirements and reviewing those already triggered. Establishing which requirements are invoked by adding a new object is a fairly fast operation, involving a traversal of the formalised guidelines. Inevitably this takes longer as more guidelines are formalised, but the use of the hypotheses as context to filter for both relevance to the new object and the current design, is effective. Reviewing outstanding requirements is also a matter of running down a list. It could be made more efficient by recording and re-using previously established facts which are currently re-proved. Locational information is an obvious example.

The real problem is the number of objects present in a design. As the number of objects present in the design increases, so does the number of candidate values for each variable, causing each proof attempt to take longer. Since we rely heavily not just on proof, but on exhaustive proof, design size will become problematic unless a mechanism is developed to focus each proof on a subset of the available objects. So far, dealing with small examples, we have not needed such a mechanism.

9 Conclusion

This paper describes an automated reasoning assistant for interactive design. For a given partial design, as components are added, the requirements assistant uses automated reasoning to:

- support designers in formalisation and interpretation
- find relevant requirements to be proved;
- check automatically whether they are satisfied;
- record those parts of the design on which requirement satisfaction depends;
- record the requirements to which components contribute;
- record the partial and complete proofs of requirements;
- make the proofs and the text they represent inspectable;
- suggest why failures to satisfy requirements occur; and
- sometimes offer a suitable remedy.

The schema-based system can capture endorsements or proof contributions without total proof each time. It is a good compromise between proof and engineer endorsement. Effectively, new lemmas are added to the theory when a component is added and instantiated.

We have taken advantage of the fact that our domain uses guidelines which encourage standardisation and simplicity. While the restrictions we have imposed in order to achieve termination and completeness in the proof system might seem oppressive, our experiments show that they have not turned out to be so. This is especially true in the safety-critical domain, where simplicity is a virtue.

Our approach relies on the formalisation of standards, requirements statements (both graphically and textually), components and combinations of components.

By describing all the potential components as objects with properties described in frames, it becomes possible to unify component and schema based approaches within an interactive framework.

Appendix

In addition to the requirements shown in the main body of the paper, we have formalised other parts of two codes of practice. The results are summarised in the following table. The variation in numbers of rules per page can be accounted for largely by the amount of descriptive background material present.

Source	Section	Pages	Rules	Title
Shell 93	4	3	43	Manual Alarms and Executive Actions
Shell 93	5	11	45	Fire Detection and Annunciation
Shell 92	4.1	4+table	96	Design Criteria for Emergency Shutdown Levels

Many of the formalisations are quite similar to each other, so we have selected a representative sample. A glossary appears at the end of the set of examples. We use ellipsis frequently (here, though not in the RA) where the original text contains a list of similar items from which we are showing an exemplar.

Example text fragments and their formalisations from section 4 of Shell 93:

"GPA call points should also be located in the following areas: ...., in crane cabs."

```
[crane_cab(A)]
        >- exists g:gpa_call_point(g) & location(g, within(A))
```

"Activation of each GPA call point shall initiate the following automatic actions: ...., if the GPA call point is in the drilling module, an individual visual and audible alarm at the drilling repeater panel, indicating the zone of the GPA call point;"

```
[gpa_call_point(G), drilling_module(M), location(G, within(M)),
 drilling_repeater_panel(P, M), visual_alarm(V), location(V, on(P))]
        >-activation(G)->>activation(V) & indicates_zone(V, M)

[gpa_call_point(G), drilling_module(M), location(G, within(M)),
 drilling_repeater_panel(P, M), audible_alarm(A), location(A, on(P))]
        >-activation(G)->>activation(A)
```

Example text fragments and their formalisations from section 5 of Shell 93:

"Many solid materials smoulder for a period before developing flames, in which case a smoke detector can give early warning of such smouldering combustion."

This is an instance of a requirement whose status is noted as being preferred rather than mandatory.

```
[solid_fire_source(S), area(A), location(S, within(A))]
        >-exists s:smoke_sensor(s) & location(s, within(A))
```

"Upon activation of a single (smoke (see note), flame or heat) fire detector, a visual and audible alarm, clearly indicating the affected zone, shall be initiated at: ... the turbine repeater panel, if the zone includes a turbine with a repeater panel."

```
[fire_protection(P), area(A), location(P, within(A)), turbine(T),
 location(T, within(A)), control_panel(turbine_repeater, A, C),
 visual_alarm(V), controls(P, V), indicates_zone(V, A), location(V, on(C))]
        >- activation(P) ->> activation(V)

[fire_protection(P), area(A), location(P, within(A)), turbine(T),
 location(T, within(A)), control_panel(drilling_repeater, A, C)]
```

```
          >- exists a:audible_alarm(a) & controls(P, a) & location(a, on(C))

    [sf_sensor(S), area(A), location(S, within(A)), turbine(T),
     location(T, within(A)), control_panel(drilling_repeater, A, C),
     audible_alarm(H), controls(S, H), location(H, on(C))]
          >- activation(S) ->> activation(H)
```

Example text fragments and their formalisations from section 4.1 of Shell 92:

"Manual shutdown facilities, interfacing to the installation emergency shutdown and process and utility shutdown systems, shall be provided both locally at the equipment and remotely at the ICC, emergency control point (TR) and at the OIM's Lifeboat station. Manual initiation shall be by dedicated push-button allowing action by a single operator."

```
    [tps_button(B)]
          >- exists a: location(B, at(a))
                     & (icc(a) or ecp(a) or lifeboat_station(a))

    [input_dev(B), activation(B)->>initiation(tps)]
          >- button(B) & dedicated(B)

    [icc(I)]
          >- exists b   tps_button(b) & location(b,

    [ecp(E)]
          >- exists b : tps_button(b) & location(b, at(E))

    [lifeboat_station(L)]
          > -exists b : tps_button(b) & location(b,
```

"isolation of the installation from subsea pipelines by closure of riser valves ..."

```
    [riser_valve(V), installation(I), subsea_pipeline(P), location(V, on(P)),
      tps_button(B)]
          >-   err_signal(B) ->> closure(V)
             & closure(V) ->> isolation(I, P)
```

"SPS shall be initiated by: ..., manual operation of a SPS push-button. SPS buttons shall be located adjacent to GPA buttons in process areas (Ref. EA/005), at the ICC and emergency control point (TR);"

```
    [button(B), activation(B)->>initiation(sps)]
          >- manual(B)
```

```
[process_area(A)]
        >- exists b :  button(b) & location(b, at(A))
                    & activation(b) ->> initiation(sps)
                       & (exists g : gpa_call_point(g) & location(g, at(A))
                                   & adjacent(b, g))

[icc(I)]
        >- exists b : button(b) & location(b, at(I))
                    & activation(b) ->> initiation(sps)
                    & (exists g : gpa_call_point(g) & location(g, at(I))
                                 & adjacent(b, g))

[ecp(E)]
        >- exists b : button(b) & location(b, at(E))
                    & activation(b) ->> initiation(sps))
                    & (exists g : gpa_call_point(g) & location(g, at(E))
                                 & adjacent(b, g))
```

SPS shall be initiated by ... confirmed fire in a hazardous area; ...

```
[fire_sensor(S), hazardous_area(A), location(S, at(A))]
        >- confirmed_fire(A, S) ->> initiation(sps)
```

Actions occurring as a result of an SPS shall be: ... Isolation from the reservoir by closure of all wellhead shutdown valves.

```
([installation(I), reservoir(R)]
        >- initiation(sps) ->> isolation(I, R)
```

Glossary

adjacent(B,G)	B and G are adjacent
activation(B)	the activation of B
area(A)	A is an area
audible_alarm(A)	A is an audible alarm
button(B)	B is a button
closure(V)	V is closed automatically
confirmed_fire(Z, S)	a confirmed fire in Z indicated by sources S
controls(S, H)	S controls H
control_panel(D, A, C)	P is a type T control panel for area A
crane_cab(A)	A is a crane cab
dedicated(B)	D is dedicated
drilling_module(M)	M is a drilling module
drilling_repeater_panel(P, M)	P is a drilling repeater control panel for area M
ecp(E)	E is an emergency control point
err_signal(B)	an error signal from B

fire_protection(P)	P is a fire protection system
fire_sensor(S)	S is a fire sensor
GPA	General Platform Alarm
gpa_call_point(G)	General Platform Alarm call point
hazardous_area(A)	A is a hazardous area
ICC	Installation Control Centre
icc(I)	I is an installation control centre
indicates_zone(V, A)	alarm V indicates zone A
initiation(X)	the initiation of X
input_dev(B)	B is an input device
installation(I)	I is an installation
isolation(I, R)	I becomes isolated from R
lifeboat_station(L)	L is a lifeboat station
location(S, at(A))	S is located at A
location(S, on(A))	S is located on A
location(S, within(A))	S is located within A
manual(B)	B is manual
OIM	Offshore Installation Manager
process_area(A)	A is a process area
reservoir(R)	R is a reservoir
riser_valve(V)	V is a riser valve
sf_sensor(S)	S is a smoke or flame sensor
smoke_sensor(S)	S is a smoke sensor
solid_fire_source(S)	S is a solid fire source
SPS	Surface Process Shutdown
subsea_pipeline(P)	P is a subsea pipeline
TPS	Total Process Shutdown
tps_button(B)	B is a total process shutdown button
TR	Temporary Refuge
turbine(T)	T is a turbine
visual_alarm(V)	V is a visual alarm

References

Clocksin 94 — Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*. 4th edition. Springer Verlag 1994.

HSE 93 — *Offshore Installations: Guidance on design, construction and certification*. HMSO 1993.

Pfleeger 95 — Pfleeger, S.L. and Hatton, L. *How do formal methods affect code quality?* to be published in IEEE Computer 1996

Robertson 96 — *Domain Specific Problem Description* in the proceedings of the *8th International Conference on Software Engineering and Knowledge Engineering, 1996.*

Shell 92 — *Emergency Shutdown and Process Trip Systems*. Shell Expro internal Code of Practice 1992.

Shell 93 *Fire and Gas Detection and Alarm Systems for Offshore Installations*. Shell Expro internal Code of Practice 1993.