Lightweight Formalisation in Support of Requirements Engineering

[1]**Jane Hesketh,** [1]**David Robertson,** [2]**Norbert Fuchs,** [1]**Alan Bundy**

[1] Department of Artificial Intelligence, University of Edinburgh
[2] Department of Computer Science, University of Zurich

Abstract

Formal design supported by automated reasoning can help keep track of requirements - a particular problem for large, detailed systems. Designers of system specifications are often constrained by codes of practice and must show not only that these have been observed 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 a "lightweight" approach, where formal proofs are used to support rather than guarantee adherence to requirements, we have developed an interactive system for formalising and managing information in codes of practice from the offshore oil industry. 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 may be useful in checking the validity of the design. It may also provide useful reference material during subsequent system modification by drawing attention to the implications that changes will have on key parts of the codes of practice.

Keywords: Automated Reasoning, Requirements, Design, Safety-Critical Systems, Software Engineering, Specifications

1 Introduction

It is a popular myth that to address complex software engineering problems using formal methods we must employ "heavyweight" methods which are sufficiently expressive to represent the full complexity of the problem. This is an exacting job which seems particularly daunting in the early stages of design, where we may not fully understand the problem to be solved. Ironically, it is in early design that anchor points for debate are most necessary. We suggest that many of the structures and methods used in formal methods can provide useful frameworks for controlling early design, without requiring us to represent formally every relevant aspect of the problem. This is done by using highly restricted formal languages, directed at clearly identifiable parts of existing design practice and supporting rather than supplanting the intuitions of designers. We refer to such applications of logic, where formal methods must be easy to pick up, as "lightweight" formal methods. A more extensive discussion of the motivation for this approach is given in (Robertson 95). In what follows we describe an application which we have tackled in this style. Our prototype system, implemented in LPA MacProlog, has been used with two central sections of Shell's "Fire and Gas detection and Alarm Systems for Offshore Installations" (Shell 93) and part of "Emergency Shutdown and Process Trip Systems" (Shell 92). The examples in this paper are based on the latter.

Safety-critical engineering relies heavily on reuse of proven designs, ensuring that designers observe best practice using standard components. A good example of this is the design of safety shutdown systems for oil production platforms by Shell ExPro. Four elements combine to shape standard practice in this domain (see Figure 1):

- The description of the physical layout of the platform. Shutdown system designers use this to identify those elements which the shutdown system must control.

- The shutdown system design documents. Shell uses 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 more detailed function block diagram, describing instruments and equipment within the modules, and their control relationships to each other. The third level, the cause-and-effect matrix, details physical indicators (such as buttons pushed or high level sensor readings) 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.

- The legislation and codes of practice set out 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 extensive and sometimes detailed. Ensuring the satisfaction of these general requirements is the responsibility of a safety engineer. Since they affect the choices made at all stages of the system lifecycle, designers must constantly be aware of them.

- The safety case for the completed system, which must be certified by the regulatory authorities and must show why the system is believed to be safe within accepted limits. The arguments in the safety case will, among other things, demonstrate that good engineering practice has been followed in satisfying appropriate legislative and company requirements.

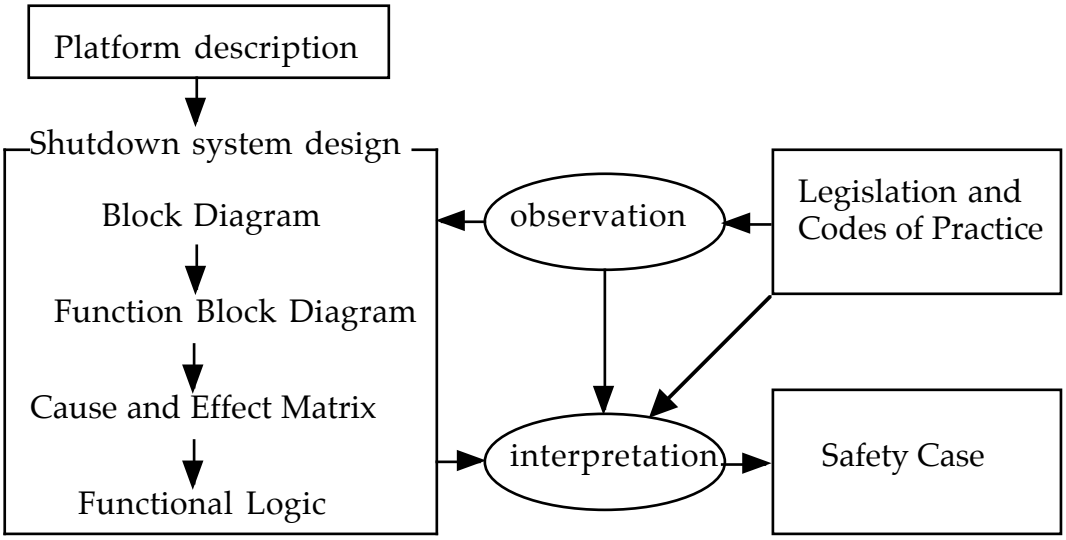


Figure 1: Overview of the design of a safety shutdown system

Two key processes connect the elements we introduced above (these are shown as ovals in Figure 1).

Observation of the requirements stipulated by legislation and codes of practice must take place when constructing the shutdown system design.

Interpretation of the requirements and design - showing how these fit together - must occur when producing the safety case.

Notice the arrow from "observation" to "interpretation" in Figure 1. This emphasises the need for synergy between observation and interpretation: if we keep careful records of the key points at which the requirements influenced design then it should be easier to show how these interact when arguing for system safety. In order to do this 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.

These activities take place in current practice but those which are observational (1 to 5) are seldom made explicit or supported in a formal way. It follows that the interpretational activities (6 to 8) are normally indirect so arguments for the safety of the system often reconstruct, *post hoc*, the means by which safety requirements were met. In current practice, this is dealt with by social and organisational means - via monitoring and standardisation in the design life cycle. Although effective in general, this approach is vulnerable in some important respects. It is possible for "small" requirements to be overlooked, only to resurface unexpectedly at a later point in the life cycle. Oil production platforms also evolve over time and it is difficult when changing the

installed system to be sure that we haven't undermined a requirement to which we agreed in the original design. This paper describes our use of formal methods to make more explicit some parts of the observational activities, making it possible to use this formal information to assist in interpretation.

As a focus for our work we built a prototype system to support tasks 1 to 8 above, concentrating on the functional logic specifications since these provide the essential system specification. The earlier design documents (such as cause and effect matrices) can be viewed as devices for subdividing and summarising the functional logic - a topic to which we return in Section 6.2. The main components of the system are illustrated in Figure 2, with rounded rectangles enclosing the support tools and rectangles enclosing the data used by and/or created by those tools.

- The markup tool is used to mark segments of the textual code of practice and to link these to corresponding formal interpretations in a logic. This task must be done prior to design of the safety system, by an engineer who understands the domain and who has some training in knowledge representation.
- The description tool is used to supply key structural features of the production platform (illustrated as $object_1$,...,$object_n$ in Figure 2) and, in doing so, classifies each of these according to a pre-specified lattice of sorts of features.
- The functional logic editor builds functional logic specifications graphically, using a library of standard schematic components. Designers work by choosing a generic schema appropriate to the segment of the shutdown system on which they are focusing; then supplying information necessary to instantiate the schema, in response to prompts from the system. Each schema carries with it a representation of key properties which it will contribute to the design (illustrated as C_1,...,C_n in Figure 2). As schemata are instantiated and added to the functional logic these contributions are also instantiated and added to a pool of contributions associated with the developing design.
- The requirements manager examines the current platform description, plus contributions from schemata, and uses a theorem prover to determine those formal segments of the codes of practice which are relevant to the design. It then allows the designer to browse through relevant formal requirements, checking (using the theorem prover interactively) whether they are satisfied by the current functional logic contributions. For those which aren't satisfied, it can assist the designer in exploring failed proofs with the aim of revealing what might be done to correct them. Most importantly, it is also able to follow back from each formal requirement to the segment of text which originated it, thus establishing chains of connections from portions of the functional logic (constructed using schemata) through to relevant formal requirements and from these to appropriate parts of the code of practice.

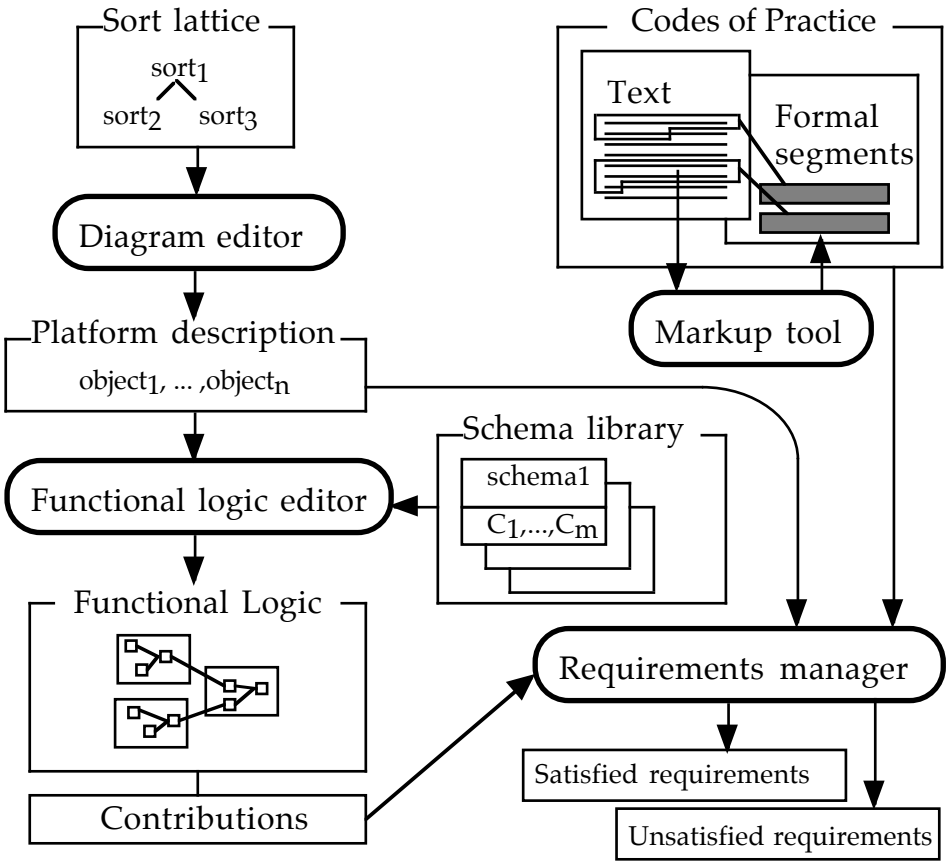


Figure 2: Architecture of the requirements management system

In what follows we shall explain the architecture of this system in more detail, explaining how we have deployed formal methods in this domain. We make no claim that our use of logic is optimal - other logical systems would certainly be as good and some might be better in particular respects. On the contrary, our aim is to demonstrate that comparatively simple applications of standard logics, carefully established in niches of the design lifecycle where they can reinforce existing practice, are both relevant and useful in early stages of design.

The structure of this paper is as follows. We describe the operation of the design support system by giving in Section 2 a scenario of existing (unsupported) design and then, in Section 3, describing how this scenario is played in our system. Fundamental to our approach is the idea that it is useful to have a formal interpretation of segments of the codes of practice and that we can provide simple tools to support this. Section 4 focuses on this issue. Section 5 then looks at the system from the perspective of each of the eight key activities which we itemised above, enabling us to discuss parts of it in greater depth. Section 6 considers some deficiencies of the original prototype and describes implemented solutions to these, whilst Section 7 identifies areas of future work. Section 8 concludes with some speculative comments about the niche which these types of lightweight method might occupy in safety engineering.

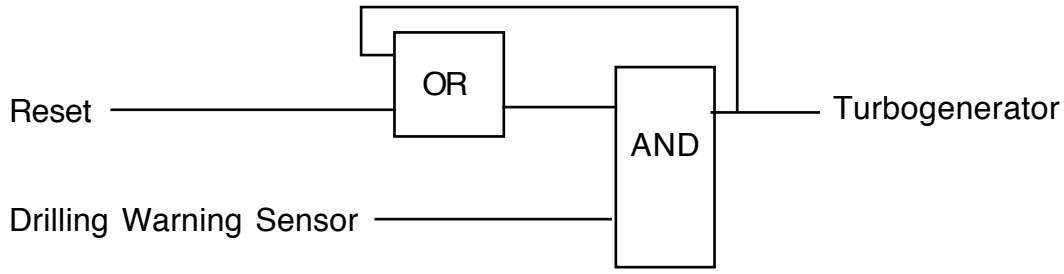
2 Scenario of Existing Practice

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. We are assuming that (as normal) 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 (we shall return to in later formal examples):

> *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 the introduction, we want to support the designer in choosing it and explaining why it satisfied the requirements. A considerable amount of work is involved in typical design which will contain 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 Formal Support for the Example Scenario

Having introduced the example scenario we now show how our system deals with it. To give the human intuition behind our technical descriptions we include in italics at the beginning of each stage of design (in Sections 3.1 to 3.3) a short summary of the main story. Readers may wish to refer back Figure 2 in order to connect the tools we describe to our overview of the system's architecture. The only part of the system not touched by this example is the markup tool, which is responsible for constructing formal interpretations of segments of the codes of practice. This is reserved for Section 4, where we discuss formalisation in greater detail. However, we need to provide some formal notation at this stage, in order to explain how requirements G1 to G3 from Section 2 are treated by the system.

In the language chosen for our system, G1, G2 and G3 are supported by four formalised requirements, R1 to R4, which are presented below with their English translations. We use a sequent based notation in which the symbol `|–' links the description of the context in which a requirement holds (on the left) with the conclusion of the requirement (on the right). To denote causation, an infix connective `->>' is used, and we use the symbol `&' to represent conjunction. Expressions of the form S = {X | C} denote the set, S, of elements, X, generated by condition, C. We assume that this will not be satisfied unless S is non-empty. The symbols `∀' and `∃' are the usual universal and existential quantifiers, with any variable appearing without a quantifier being assumed to be universally quantified across the whole expression (a standard convention in logic programming).

```
machine(X), initiator(I), expected_consequence(I,X)
|– tripped(I) ->> shut_down(X)                                    (R1)
```

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),  Bs = {B | (initiator(B) & expected_consequence(B,X))}
|– reset(X) & (∀s  (on(s,Bs) & ok_signal(s))) ->> ok_signal(X)    (R2)
```

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)                             (R3)
```

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) (R4)

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

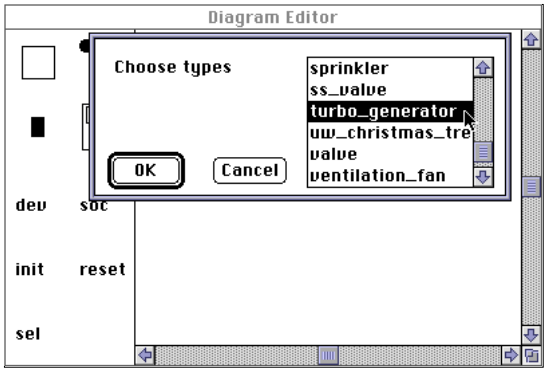
The requirements have been formalised in logic so that we can attempt to show each one to be true in each context in which it applies. The requirement will be considered to apply whenever all the elements of its context (on the left of the `|–') are satisfied from the current design. The resulting set 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 indicate the nature of what is needed.

3.1 Inserting Components using the Diagram Editor

The designer sketches part of a platform, containing a turbo generator and a drilling warning sensor and, on prompting, recognises that signals from this sensor are intended to control the generator. This information suggests that a number of segments of the code of practice may be relevant, and that these are currently not satisfied. Several of these talk about a reset facility, prompting the designer to add a reset button to the platform sketch.

Design of safety shutdown systems for real production platforms start by relating elements of the platform (such as areas, doors, equipment or escape routes) to elements of the shutdown system (such as initiators or reset controls). In our system, this process is simulated by allowing designers to specify components taken from a tool palette of choices (see below).

Selecting any of the objects from the palette allows the designer to place an item of that sort on the design window. Depending on this tool choice the designer will be presented with a menu generated from a sort lattice (described in Section 4.6) asking more specifically what sort of object this is to be:



Having chosen the type of the device, the system responds by adding a

token to the editing window, and recording that the design contains a device controller automatically labelled tg0 which is of sort "turbo_generator". The system scans the requirements, selects the two which now apply (R3 and R4 because these are the only ones for which all of the hypotheses are satisfiable), instantiates them, and on failing to prove them, adds the instantiated versions to the list of outstanding requirements. Had they been provable they would have been added to the list of satisfied requirements. The designer can inspect either of these lists at any time. Adding the initiator (the drilling warning sensor) proceeds similarly. A new component, dws0, is added and recorded as being of sort "drilling_warning_sensor".

The existence of this new object provides a value for I which makes initiator(I) true in the context of R1. Since X and I have been instantiated in requirement R1, the last of its set of context formulae is now expected_consequence(dws0,tg0). Although the system 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. Such predicates are pre-defined to be "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. All queries of the designer are made using a simple domain-specific paraphraser which generates an English description of the formal term. In our example the designer confirms that dws0 is expected to control tg0 and the system attempts to prove

machine(tg0), initiator(dws0), expected_consequence(dws0,tg0)
|– tripped(dws0) ->> shut_down(tg0) (R1')

but 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 system can generate another three outstanding requirements which are:

machine(tg0), Bs = {B | (initiator(B) & expected_consequence(B,tg0))}
|– reset(tg0) & (∀s (on(s,Bs) & ok_signal(s))) ->> ok_signal(tg0) (R2')

machine(tg0) |– ∃ r local_reset(r) & resets(r,tg0) (R3')

machine(tg0) |– shut_down(tg0) & not_reset(tg0) ->> latched(tg0) (R4')

The designer now adds a reset button to the design, in a similar way 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 the 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. When the

reset is inserted, the designer's menu choice will be recorded with it. From this information, definitions (see Section 4.5) allow the system 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 automatically.

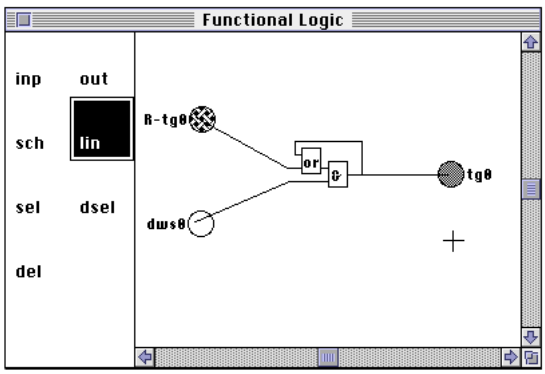
Lacking the connecting circuitry, the system cannot complete proofs at this stage. On request however, it can advise the designer by trying to look one step ahead and see if any known component would be useful, as we describe later.

3.2 Inserting Components using the Functional Logic Editor

The reason why many of the requirements stipulated by the code of practice cannot be confirmed is that there is no functional logic to account for the appropriate system behaviour. Using a library of standard structural forms, the designer constructs the functional logic connecting reset and sensor to an output controlling the generator.

The designer switches to the functional logic editor by a menu bar selection. The system displays a new window with a tool panel which allows the insertion of inputs, outputs and schemas, again prompted by menus of alternatives (see display below). Although the functional logic editor operates in the same context as the design editor of the previous section there is no assumption that all the components present there will be involved in this logic. The designer must choose an appropriate subset of them. When the designer selects **inp** from the tool panel, the system generates all possible inputs (dws0 and R-tg0) and 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 latched reset from the menu places a diagrammatic representation of one on the window (see below); adds a named instance of it to the formal description (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.



The contributions added by this schema are as follows (Section 4.4 shows

the details of the schema responsible for generating these):

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

Notice the lightweight nature of these contributions. They do not define the functioning of the shutdown system in full detail, since (for example) they do not describe exactly what happens when latching occurs as a consequence of some stimulus. By describing the contributions of an object or logic circuitry at a high level, details of its temporal properties and how they are achieved can be finessed while preserving essential behavioural information. For example the system'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 detail. It is essential that we do this because it would be prohibitively time consuming to model all potential behaviours of the shutdown system, whilst the resulting complex model would be difficult to communicate to designers. The price we pay is that we cannot guarantee observance of requirements - we can only suggest which might be relevant and help the designer check observance based on pre-defined properties of schemata.

3.3 Completing Requirement Proofs

The structure provided by the functional logic is sufficient to suggest that the requirements flagged as relevant during early design are now accommodated. However, this is not a cast-iron guarantee. The designer steps through the argument for satisfaction of some of the requirements to see whether they make sense to her.

In addition to completing the design, the designer has now satisfied all the requirements. That is, for every requirement, C |– R, if we can find an instance for C in the current design then we can prove R in the design. These instances in the current example are R1' to R4' in Section 3.1.

The proofs of all the requirements apart from R3' are trivial because their conclusions match schema contributions directly, modulo renaming of variables. Proofs of satisfaction of these requirements 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)  |– ∃ 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 meta-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

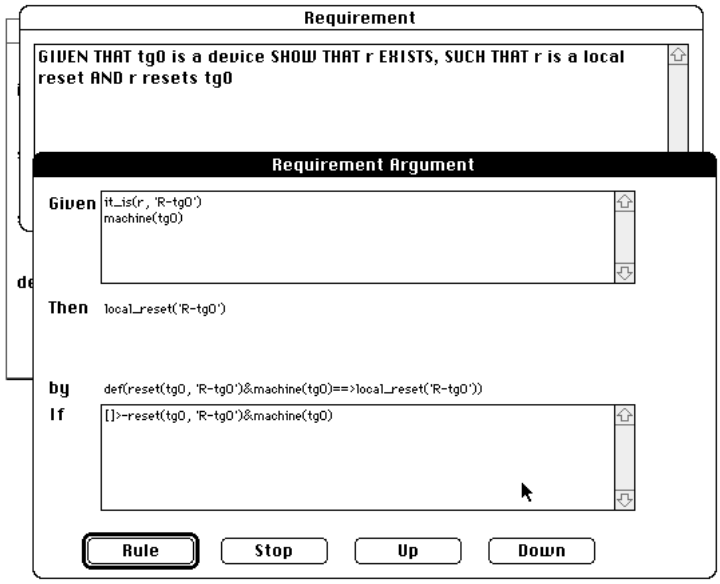
proof rule for ∃. The prover attempts to prove:

machine(tg0) |– local_reset(R) & resets(R, tg0)

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

machine(tg0) |– local_reset(R)
machine(tg0) |– 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. The screen snapshot below shows the system's reconstruction of this part of the proof. The requirement is translated to pseudo-English, whilst the details of the proof are displayed in more conventional logical notation.



The prover uses the definition of local_reset (given in Section 4.5) 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 system by piecing together information from the designer's explicit additions to the design and by inference from contributions due to some of these choices.

4 Supporting Formalisation

There is always a tension between the desire for sophisticated formal representation and the need for clear communication to non-logicians. As a compromise, we chose to work with a lightweight restricted language, accepting 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. In this section we introduce our source material: Shell's textual codes of practice. We then justify our choice of formal language; explain how we support formal interpretation; and describe three types of structure (schemata, definitions and sorts) which are used to connect designs to formal requirements.

4.1 Our Starting Point: The Shell Code of Practice

Shell's internal code of practice (Shell 92, Shell 93) 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 prescriptive. The guidelines change infrequently after careful checking, reducing the likelihood of radical re-interpretation of the text. They are written in clear simple language using regular well-understood terms, which lend themselves to formalisation using equivalent terms to those used in the text. By arranging all the objects described in the codes of practice in a sort lattice (see Section 4.6) we encourage the use of standard classes of objects. It also helped that standard operations such as tripping, shutting down and isolation were obvious. Consistency of terminology in the original source text is crucial to this sort of description. Otherwise there is a danger that, in an effort to minimise the number of predicate names in our formal interpretation, we record as similar cases which are significantly different. Alternatively a confusing plethora of descriptions may be introduced covering subtle but unexplained complexities - the disadvantages of a deeper formalisation without the advantages. Either of these problems is a likely source of miscommunication.

Safety engineers using the markup editor should never claim that the formal translation they have chosen is a precise expression of the corresponding natural language. The formal representation, and the automated reasoning which it supports, is a vehicle for alerting designers to parts of the textual code of practice which are relevant in order to make informed decisions. 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 normally associated with deeper formalisation.

4.2 Choice of Formal Language

As a "lowest common denominator" logic we chose a restricted first order predicate calculus with sets and including a causation operator ('->>'). The strategy used in satisfying requirements resembles that of logic programming languages, like Prolog, enhanced with facilities for accessing the components of our design and for questioning designers. However, we have to ensure that our proofs always terminate, since our theorem prover frequently searches exhaustively for a solution. This is ensured by the following restrictions on our representation and inference:

- All the basic data, forming the leaves of any proof tree, are required to be ground (*i.e.* the formal expressions contain no logical variables). We can be

sure of this because information is only available to the prover from the following sources:

- ♦ Objects which are grounded in a platform diagram.
- ♦ Properties contributed by objects in the design, via rules which we have checked can only generate a finite set of ground instances.
- ♦ Sort information and axioms, supplied by the designer via an interface which tests for groundedness before accepting the information.
- ♦ Design endorsements, which are logical constants and therefore ground.

- Implications are not allowed to introduce new variables, so each will always conclude a finite set of ground expressions from ground preconditions.
- Recursive definitions are not allowed, so non-terminating recursions aren't possible.
- The prover will only attempt to satisfy goals which are either single goals; conjunctions; or disjunctions. The solution to any goal consists of findign an instance which satisfies it. No other forms of goal are allowed.
- Conjunctive goals are always proved by finding an instance satisfying all of the goals, so if each goal is terminating so is the conjunction.
- Disjunctive goals are always proved by finding an instance satisfying any one of the goals, so if each goal is terminating so is the disjunction.
- We use the closed world assumption to allow the negation of a ground expression to be concluded only if there is no available proof for the expression, thus if all proofs of positive goals terminate so do proofs of negative goals.

Some of the information needed to establish requirements concerns spatial relationships (for example, that a particular alert button should be within a control room). In our prototype it is convenient to express these graphically in a diagram editor, with the system then applying special-purpose procedures to infer the appropriate spatial relationships. Those currently supported include: *within, at* and *leads to* relationships.

4.3 Supporting Formal Interpretation

The system provides a markup editor (see Figure 2) allowing an appropriately trained safety engineer to read an English text online, mark phrases or sentences, and then record his or her translation of them in our formal language. 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, so files of formalised requirements can be saved and subsequently loaded by shutdown system designers.

Producing formal interpretations for parts of the codes of practice cannot be done without some experience in knowledge representation. Our system assists in this task but it does not (arguably, cannot) relieve authors of the formal requirements from responsibility for choosing appropriate 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:

device(D) & passed_error_threshold(D, S) ->> 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 (for example, suggesting causal relationships) provided that each predicate is well-defined and used consistently. It is also close to a level at which the textual codes of practice are presented, and while not offering a deep analysis, it offers a broadly comprehensive one. At present, the system's support tools for terminology management are still primitive, and it relies largely on the same human discipline that engineers use now. However, we have experimented with more sophisticated support tools (see Section 6.1).

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. There will inevitably 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. We leave this decision to safety engineers, and offer them formalisation and reasoning support.

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 hypotheses (the part to the left of the `|–' symbol) are not used in quite the usual way. 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 system 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, if they are so specific that anything which matches the context already satisfies the requirement 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. To illustrate this point, consider requirement R3:

machine(X) |– ∃ r local_reset(r) & resets(r, X) (R3)

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, R3 is invoked as soon as there is a device X (machine(X)), and reminds the designer that a local reset will be needed (∃ 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) |– resets(R, X) (R3º)

R3º says that if there is a device X (machine(X)), and an associated local reset for X (local_reset(R,X)) this should reset X (resets(R, X)). The two formulations are clearly not equivalent, but they are both possible partial translations of G3. R3º 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.

4.4 Design Schemata and their Contributions

The building blocks of the functional logic are schemata. 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 from the current design 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, 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 this as a contribution, rather than requiring it to be inferred from its internal logic. Instantiation of a schema's context and contributions is done in a similar way to that for requirements, so each single contribution can generate multiple contribution instances, corresponding to each way that its hypotheses can be true. When a schema is added to the design, the system instantiates and records the contributions it makes. 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)
               Is = {X | (initiator(X) & expected_consequence(X,O))}
               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 standardisation which already exists in this

domain. Engineers and designers working on designs and equipment recognise standard configurations, and schemata save them from having to formulate similar contributions every time the same sort of configuration 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.

4.5 Definitions

Some of the terms used in the codes of practice are convenient shorthands for more complex formal expressions, to which we will want access during theorem proving. To allow this we supply definitions which 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. 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 used in Section 3.3 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 an operations control centre:

```
reset(X) & machine(X) ==> 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.

4.6 The Sort Lattice

Designers and safety engineers will routinely refer to classes of objects such as `initiators' or `machines'. In order to enable the same kinds of reasoning, the system uses a simple, non-parameterised sort lattice. To complete the details of our running example, we include the appropriate segment of this in Figure 3 (with sorts to the left including those to the right).

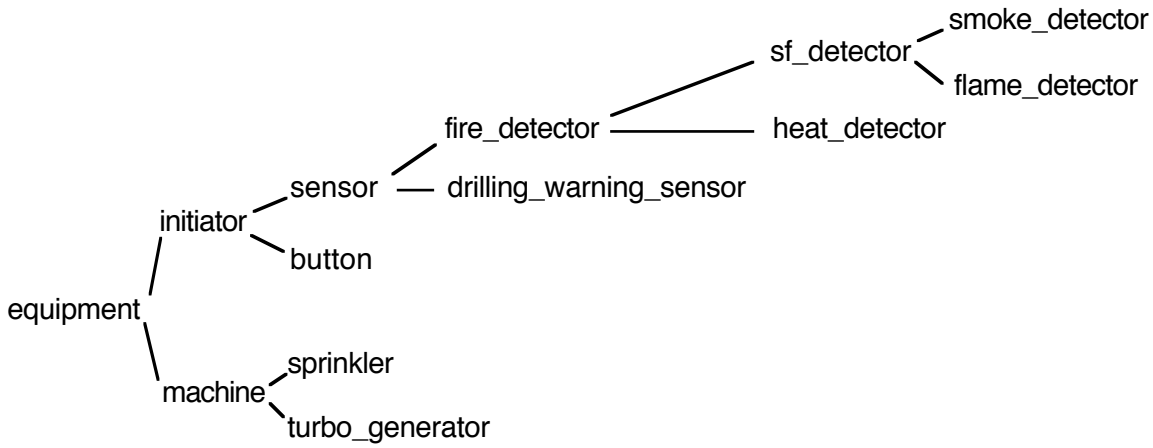


Figure 3: A segment of the sort lattice

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 unary predicates to describe them. Although a little less efficient, we then achieve the functionality of a sorted logic by explicit declarations in the 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 system then takes as relevant those requirements which have a positively occurring predicate in their context which is an element of that set (*e.g.* machine(X)).

5 Requirements Management

In the introduction, we itemised eight key requirements management tasks. Here we will explain how our system supports these tasks.

5.1 Noticing when requirements apply

Usually each component which is inserted into the design 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 system helps keep track of those requirements which are formalised, as we demonstrated in Section 3, and checks for relevant requirements incrementally, as each design step is taken. In case later context elements can provide information which might make it possible to establish earlier ones, the system 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 the satisfaction of others, this process is made considerably more efficient. However, there remains a high computational overhead for this algorithm. We explain in Section 6.2 how modularity can be used to reduce this problem.

5.2 Keeping track of all the different requirements

In addition to letting designers load a list of applicable requirements at startup time, the system maintains lists of all the satisfied and unsatisfied instances of requirements which have been invoked. 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 system applies the requirement afresh as demanded by the current context and finds and cancels any superseded versions. This happens in the following example, which uses requirement R2 from our running example:

machine(X), Bs = {B | (initiator(B) & expected_consequence(B,X))}
|– reset(X) & (∀s (on(s,Bs) & ok_signal(s))) ->> ok_signal(X) (R2)

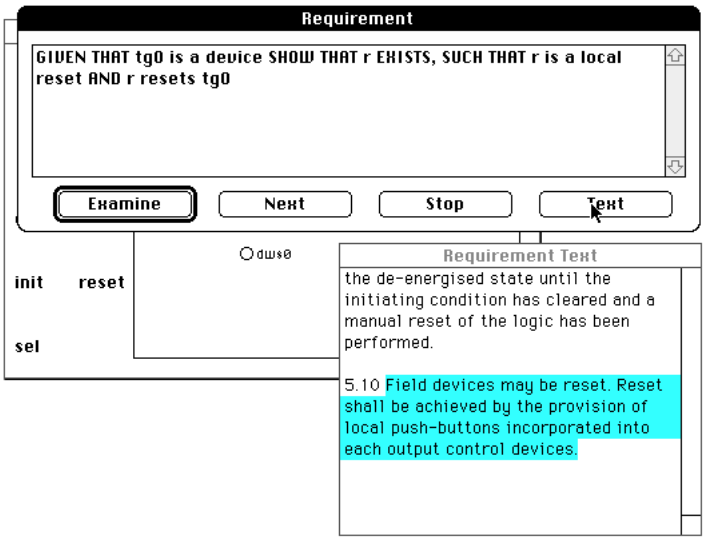
which describes the property that a device controller which has been reset should have an OK-to-function signal when all its sensors are also signalling OK-to-function. When only one initiator is present, say a drilling warning sensor, dws0, Bs is just the set {dws0}. But if another initiator, a flame 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.

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 would 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. In other words, for requirement instance A to be detected as possibly superseding requirement instance B there must be at least one element of the context of A which is the same as that of B and at least one element of the context of A which is different from that of B.

5.3 Assisting with requirements' interpretation in the current context

At any time, the designer may review the state of the requirements, whether satisfied or outstanding. The system shows a simple pseudo-English translation of the formal requirement by default - although for some tasks it is still necessary to read formal statements directly. 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.

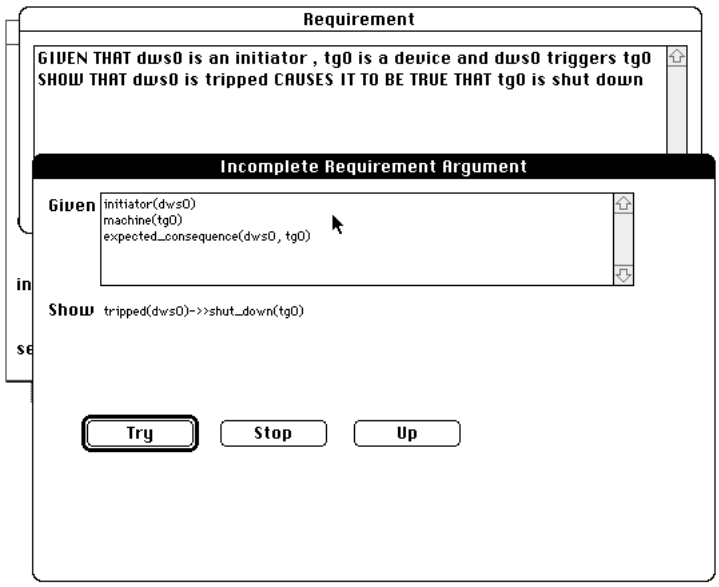
The next figure shows the designer viewing invoked requirement R3'. The designer has opted to check on the original text too, by clicking **Text**. The highlighted part is the original text from which the formalisation was obtained and it appears in the context of the original document.



Translation into pseudo-English is accomplished by means of a Definite Clause Grammar (a standard language translation facility in Prolog, described in Clocksin 94 and elsewhere), 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**.

5.4 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.



5.5 Proposing remedial design for unsatisfied requirements

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 system's database. The system can offer advice on what to do. For a given outstanding requirement, the system can use abduction to see if the addition of one of the known components has the potential to complete the proof. In the example, while looking at the details of R1', the designer might ask for such advice:

The advice is calculated by looking at the applicable schemas' contributions (see Section 4.4) 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 Section 4.4 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.

5.6 Evidence of requirements' satisfaction

The evidence is the proof which the system has automatically constructed using the facts supplied by the designer's addition of objects and their contributions and relationships (see Section 3.3).

5.7 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.

5.8 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.

6 Extensions to the Basic System

The system described in the preceding sections is a prototype and from our experience in building it we found various areas in which it could be improved. Two of these seem fundamental: the need for greater support in marking up the codes of practice; and the need to limit the focus of attention in design. Our investigations of these are summarised in Sections 6.1 and 6.2. Section 6.3 discusses a third area of further research which is, as yet, unexplored: the development of mechanisms to support non-monotonic evolution of the design.

6.1 Structured Editing of Formal Expressions

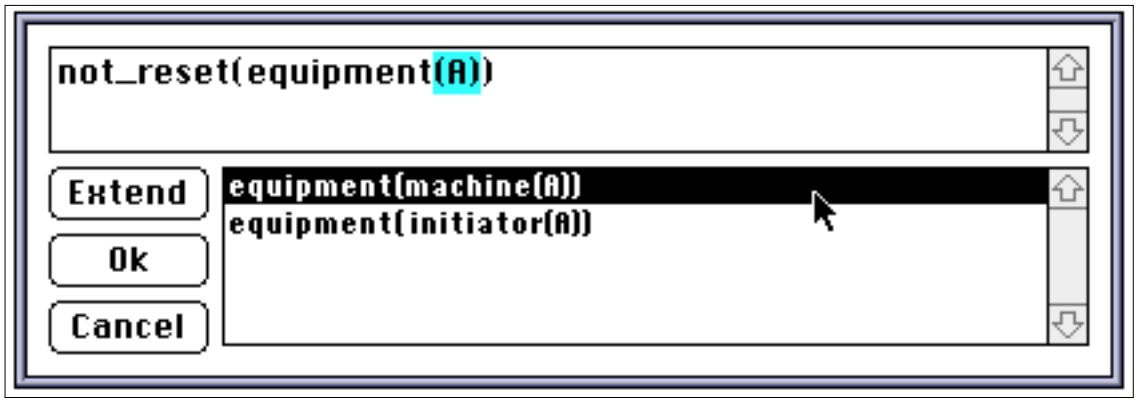
The original markup editor (Section 4.3) links formal expressions to the text of a code of practice but it gives little assistance in choosing the vocabulary or obeying the grammar of the formal language. Such assistance is necessary because people doing markup are unlikely to have much training in the

mechanics of formal methods and it would be helpful if the system could emphasise the appropriate structural forms for expressions. More importantly, we would like to minimise the danger that different words are used by accident for the same concept. It would be useful if the markup editor could promote an awareness of the terminology currently in use and require the engineer to do something special in order to extend the existing vocabulary. Of course, this doesn't prevent misuse of terminology but it gives the issue the prominence which it deserves.

Our extension to the markup editor allows formal expressions to be composed by selecting from a library of template predicates; restricting the sorts to which those predicates apply; and finally combining these predicates using the standard logical connectives. As an example let us consider formal expression R4 from Section 3 which, for convenience, we repeat here:

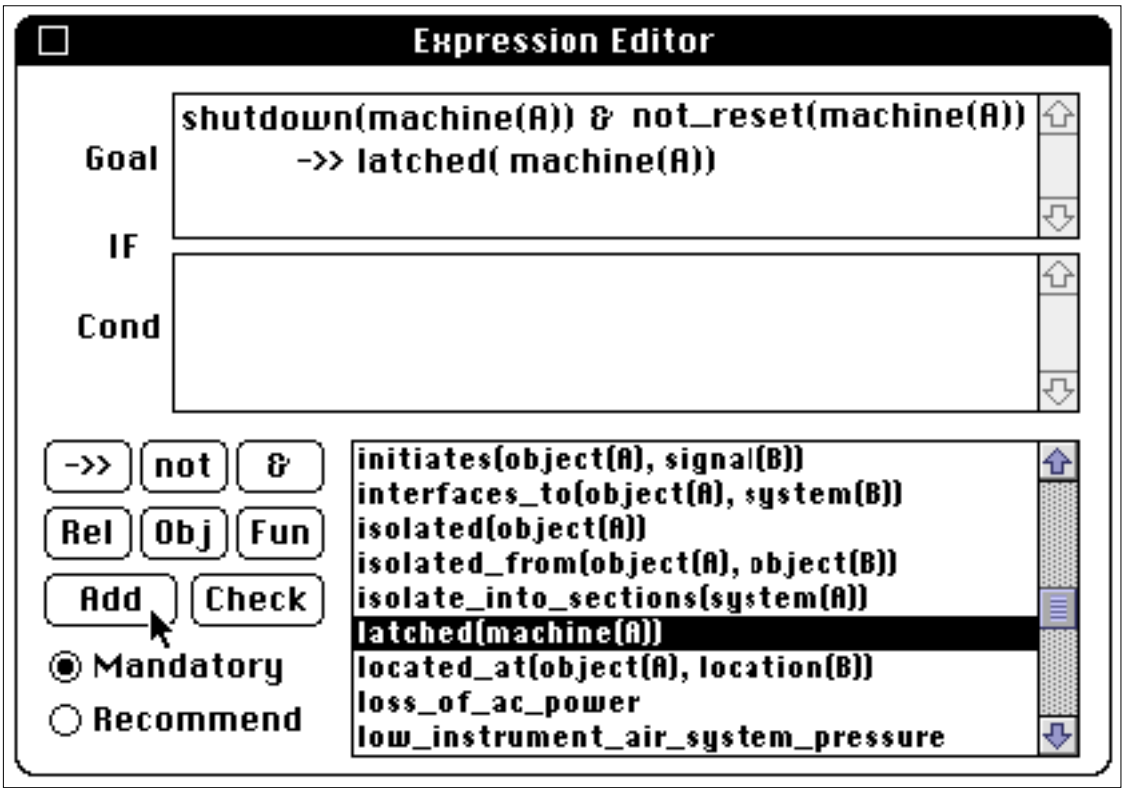
```
machine(X)
|– shut_down(X) & not_reset(X) ->> latched(X)                    (R4)
```

Our structure editor allows this sort of expression to be constructed by first describing each of the predicates which it contains. Suppose we want to describe the predicate named not_reset. The editor offers a menu of currently recognised predicate names, along with the most general sort restrictions on their arguments. One of these is not_reset(equipment(A)), where the term equipment(A) denotes that the argument to this predicate must be some sort of equipment. If we select this template we are prompted to restrict the sort of the argument, via the display below. Here we see in the top box the expression we are restricting and, below it, a menu of possible restrictions on the variable, A. There are two of these (equipment can be restricted to machine or initiator), corresponding to the subsort relations from Figure 3. In this case we would select the first option, restricting not_reset to apply only to machines in this expression.



By repeating a similar interaction for shutdown and for latched, then connecting the restricted predicates by appropriate logical connectives, we obtain the expression shown in the top box of the display below. This shows the requirement goal at the top with any additional contextual conditions in the box below (these would be on the right and left, respectively, of the `|–' operator). The menu at bottom right allows selection of predicate templates from the

library, as we described above), whilst the buttons perform standard operations like adding logical connectives or creating new templates. The radio buttons at bottom left allow the requirement to be labelled as either mandatory or (less strictly) recommended.



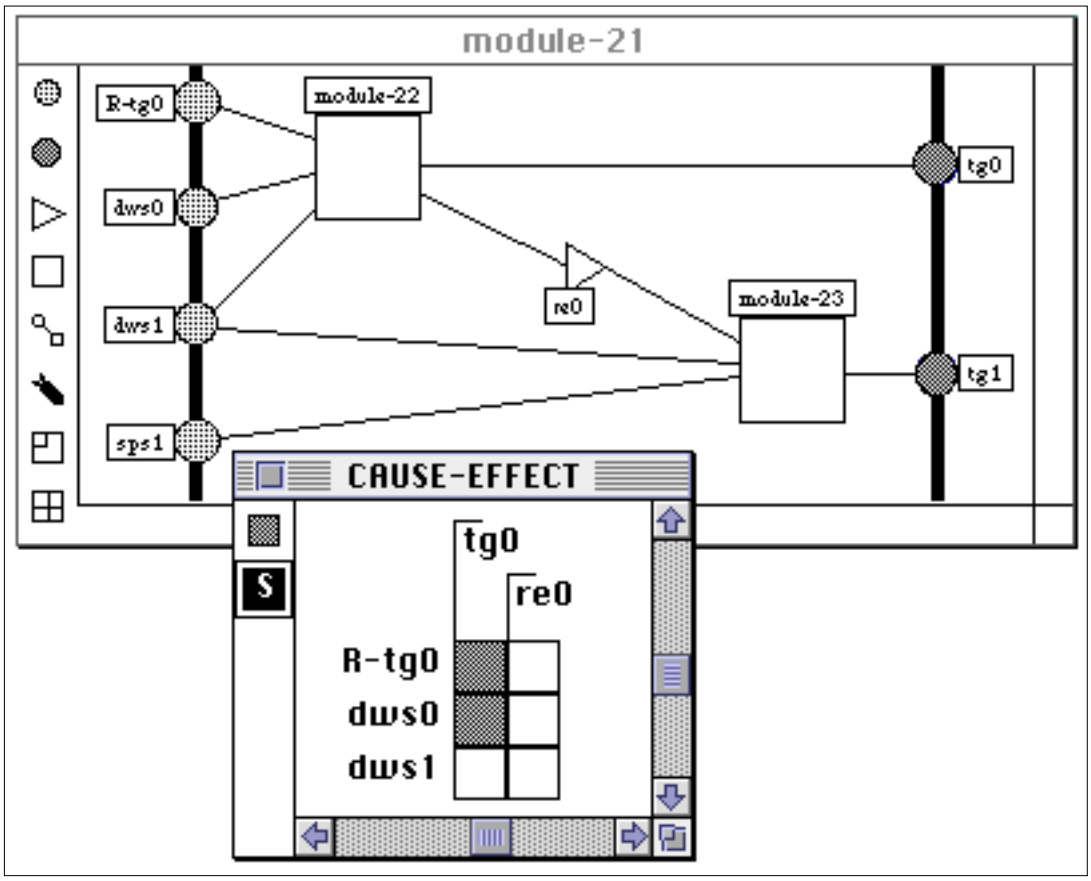
The formal expression in the display above isn't quite the same as R4 because machine(X) doesn't appear in the condition box. However, it is straightforward to infer this from the machine(A) sort restrictions in the goal. The appropriate general transformation is that each sort restriction of the form S(A) in the goal can be replaced by a new variable, X, and the sort restriction added as a new condition, S(X). This gives us R4.

6.2 Focusing on Design Modules

A typical safety shutdown system for an entire oil production platform has over a thousand inputs (sensors, buttons, *etc*.) and roughly the same number of outputs (alarms, trips, *etc*.). Our restricted theorem proving strategy (Section 4.2) gives us a finite search space but this would be very large for a full platform design. There are two ways of reducing the search space without changing the existing proof strategies. One is to be selective about the requirements which we attempt to prove. The original system allows selectivity in exploring proofs (Sections 5.4 and 5.5) but searches exhaustively when determining the relevance of requirements (as in the example of Section 3.1). This is essential in order to avoid "blind spots" - areas of the code of practice which could have been detected as relevant but which were ignored due to operator error. The other way of reducing the search space is to partition the input/output space in some way - only applying the theorem prover to those inputs and outputs which are

functionally related at an appropriate scale. This form of modularity wasn't supported in the original system but does feature strongly in standard practice in the domain - it is one of the main reasons for the block diagram, function block diagram and cause-effect matrix notations which we mentioned in the introduction and in Figure 1. We have experimented with a graphical interface for dividing the design into decomposable modular components. We describe this by reconstructing our running example in a modular style.

Suppose that we are designing a module of the system which takes as inputs the reset, R-tg0, and drilling warning sensor, dws0, from the example of Section 3.1 and, in addition the drilling warning sensor, dws1, and shutdown button sps1. The outputs from this module are the trip to the turbogenerator, tg0, from our example and a second trip, tg1. This module is shown diagrammatically in the screen snapshot below. The label on the frame of the window, module-21, is its name. The thick line on the left represents the input "perimeter" of the module, with the four inputs arranged along it. A similar line on the right represents the output perimeter. Inside this module we have decomposed the design into two further modules, module-22 and module-23, with an output, re0, from module-22 being input to module-23. The lines from R-tg0, dws0 and dws1 to the left side of module-22 denote that these inputs are carried through into that module. We could decompose module-22 in a similar way to module-21 but decide instead to describe its functional behaviour. On request, the system can generate for any non-decomposed module a cause-effect matrix, as shown in the small window on the diagram. Each row in the matrix is an input; each column is an output; and by marking appropriate cells the designer can indicate which of the inputs are expected to influence specific outputs. In our case, we specify that R-tg0 and dws0 will influence tg0. This specification then allows the system to set up the inputs and outputs for constructing diagrammatically the functional logic specification shown in Section 3.2.



This modular structure allows the designer to focus the attention of the requirements system in a methodical way, since requirements can be checked relevant to a particular module (where the numbers of inputs and outputs are comparatively small) rather than across the entire system at each design step. Decomposition of modules into sub-modules allows this focus to be raised to larger modules or lowered to smaller, more specific ones - thus allowing checks for requirement interactions between sub-modules.

6.3 Maintaining Consistency as Designs Evolve

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 have already been obtained may need to be reproved. As we demonstrated in Section 5.2, a proof that was true for all the sensors in an earlier partial description of the system may need to change when a new sensor is added. As described earlier, the system monitors this process and adjusts superseded requirements, relying on an algorithm which tests whether new requirements subsume existing ones. In general, design isn't necessarily monotonic - a designer could introduce some new component which undermines one of the requirements which had been satisfied earlier. A general solution to this problem is possible in two ways.

The most direct solution would be to prove all the requirements anew at each stage in the design (or give the designer the use of this facility more

selectively). This is easy to provide within the existing system but would be time consuming - although the use of a modular system like the one described in Section 6 could mitigate this objection. Perhaps more importantly, regular and complete regeneration could be difficult for engineers to monitor by eye, and they could be several steps down a design trail before realising that requirements had been overhauled in a radical way. The original method, which replaces existing requirements by those which subsume them, maintains greater continuity.

A more sophisticated solution would be to employ a truth maintenance system to manage the evidential dependencies between requirements and to flag those part which become undermined by non-monotonic design steps. There are precedents for this type of work in knowledge based systems and requirements engineering systems but merging truth maintenance with both the design and monitoring operations of our system raises some difficult technical and human-factors issues which we have not addressed.

9 Conclusion

Formal specification, performed using a logic, is often undertaken to provide conclusive guarantees of key properties of a design. This normally involves detailed and (we hope) complete modelling of appropriate aspects of the design, and from these we may prove that desired properties will always hold. The application we have described in this paper is not in this traditional mould. We model the system only superficially and our proofs do not guarantee that key functions (such as latching) will hold in the completed design. Instead we use the mechanisms of proof to connect components of the design to relevant segments of the codes of practice, via axioms contributed by schematic design components. Our proofs are not intended to guarantee properties but to present arguments for relevance of requirements to design. This allows our system 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.
- Sometimes offer a suitable remedy.

Although they are presented to engineers using graphics and “natural” language text, the mechanisms underpinning our system are those of traditional computational logic: schemata to introduce new lemmas to our theory; deduction to distinguish satisfied from unsatisfied requirements; abduction to suggest means of satisfying unsatisfied requirements. It is the interpretation placed on our proofs, not their form, which distinguishes this from traditional applications.

Acknowledgements

This work was funded under the DTI/EPSRC Safety Critical Systems Programme on grant GR/J58619 and by an EPSRC Advanced IT fellowship held by David Robertson. We are grateful for the support provided by others on GR/J58619 (Stuart Anderson, Glenn Bruns, George Cleland, Corin Gurr, Robert Inder, Donald Mackenzie, and Keith Stenning); our industrial contacts at Shell ExPro (Nick Curley) and at Adelard (Robin Bloomfield). We also thank the journal's referees for their influential comments on the original draft of this paper.

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 95 — Robertson, D. *Lightweight Specification*, in Proceedings of the ONR/ARPA/AFOSR/ARO/NSF workshop on Software Architectures, Monterey, California, 1995.

Robertson 96 — *Domain Specific Problem Description,* in 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.