Reliable and Accountable System Design

The Knowledge Engineering Review, Vol. 12:3, 1997, 289–305

Reliable and accountable system design

PAUL KRAUSE[1,3], JANE HESKETH[2] and DAVE ROBERTSON[2]

[1]*Philips Research Laboratories, Crossoak Lane, Redhill, Surrey, UK*
[2]*Department of Artificial Intelligence, Edinburgh University, 80 South Bridge, Edinburgh, Scotland*

Abstract

Few would disagree with the assertion that safe engineering starts from the early stages of system design and should be maintained throughout the lifecycle. Different engineering domains have developed, mostly informal, frameworks with which they hope to promote this attitude. An interesting question for the KBS community is whether some of our methods for knowledge representation and reasoning can be used to assist in understanding, representing and interpreting such frameworks. This paper concentrates on what is (arguably) the area of greatest concern: relating system requirements to high level design. We highlight what appear to be the major difficulties which face us in this area, using examples from systems which have been built to tackle them.

1 Introduction

It has been suggested that the majority of software failures stem from misunderstandings of the operational requirements of applications. Combined with this, we may find that arguments for the safety of software, in the context of system requirements, are poorly articulated. In this paper we discuss a number of approaches that may be taken to provide formal representations of key aspects of *system requirements* and to linking these to *safety arguments*. In particular, we are interested in studying how knowledge-based approaches may assist in:

- representing and reasoning about system requirements;
- articulating arguments for the safety of a system;
- harnessing domain knowledge to control the generation of system requirements.

We should be explicit over precisely what aspects of the software lifecycle we are aiming to support. Figure 1 outlines a fairly standard "V-model" for software development. As a high level view, this outlines the progression from requirements elicitation, through specification, detailed design and coding. Once coded, modules are tested individually, then tested further as they are integrated into the complete system. Finally, once the system is complete, system tests and then acceptance tests are performed. The horizontal lines indicate the relevant aspect of the development process that the system is being tested against in a particular test phase. For example, during the acceptance tests one is verifying that the final implemented system meets the requirements agreed in the requirements document. In the system tests, on the other hand, one is validating the implemented system against the specification.

As it stands, this provides a rather weak (in terms of poorly controlled) software development process. Of particular concern is the implication that testing is an activity that is carried out during the late stages of the process. This is indeed the case in many projects. However, this has a serious

[3]Paul Krause started work on this paper whilst a member of the Advanced Computation Laboratory, Imperial Cancer Research Fund, London UK, and gratefully acknowledges Professor John Fox for use of the department's facilities during this time.

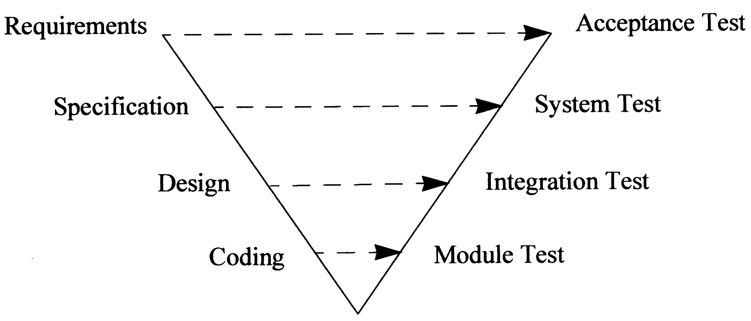

Figure 1 Standard "V-model" for the software development process

problem. Faults that are *introduced* in the early phases of the development process are often *detected* in the latest phases, at a point when they are deeply entrenched in the system design, and hence most expensive to correct. Figure 2 gives a graphical representation of this. Maisey and Dick (1996) used the idea of a defect grid to analyse the effectiveness of software development processes. They are read in an analogous way to the grids of distances between towns found in road atlases. The number in a square formed by the intersection of a "row" and a "column" is the number of faults detected at the phase labelling the row that were introduced during the phase labelling the column. For example, in the hypothetical case illustrated, seven defects detected during the acceptance tests were introduced during the requirements phase. Although this is a hypothetical case, it is based on data derived from a real project.

As mentioned, defects introduced early and detected late can be very expensive to correct. The ideal to strive for is a software development process whereby all defects are detected at the earliest stage possible. The use of structured or more formal development methods can help with this in two ways. Firstly, the product of a given phase can be verified against the product of the preceding phase (a detailed design verified against a specification, for example), so reducing the chance of introducing defects. Secondly, the generation of the product of a given phase can help validate the product of the preceding phase (the design activity may highlight inconsistencies in the specification, for example), so introducing an early test activity.

Now, although there are structured, semi-formal and formal approaches which can be followed from specification through to the later phases of the software process, the requirements elicitation phase is much less well supported. The provision of appropriate formal support is a hard problem,

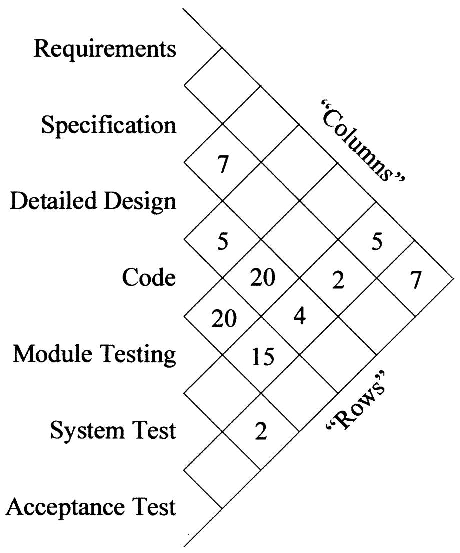

Figure 2 A hypothetical defect grid, redrawn after Maisey and Dick (1996). Note, "rows" travel from bottom left to top right, whilst "columns" travel from top left to bottom right

for reasons which will be enlarged upon in the next section. Nevertheless, if we are to achieve the goal of enabling validation and verification activities to be carried out from the early phases of a project, it is vital that this problem be addressed.

The use of the word "vital" in the previous sentence makes for a strong claim. However, recent work clearly demonstrates that additional effort invested in the requirements elicitation phase of a project has a significant payoff in the later phases of the project (Blackburn *et al.*, 1996). The savings in rework due to missing or misunderstood requirements are substantial, with additional benefits accruing through the earlier realisation of a stable functional specification and shared understanding of the project goals. In the case of object-oriented specification and design, Jacobsen's use-cases provide a tool for eliciting an initial model in terms which are readily understandable by all project stakeholders (e.g. Gough *et al.* 1995). However, as yet formal methods such as Z provide no such technique. Guides to the use of formal languages, of which Woodcock and Davis (1996) is a good example but still typical in this respect, start with a specification and then proceed through refinement and proof to an implementation. They are agnostic as to how the engineers identify what is to be specified in the first place. What is needed is support for the elicitation and analysis of requirements in a language which is communicable to *all* project stakeholders, yet is sufficiently precise that the resulting model can provide a basis for the development of a formal specification.

The goal of this paper is to stimulate further work in this area by using three examples of approaches that have been taken. Although they provide partial solutions to the problem, they are also used as a framework for discussing and illustrating some of the difficulties that remain to be addressed.

2 Components of a safety argument

In the introduction, we discussed the aim of studying how knowledge-based approaches may help in the representation of, reasoning about and generation of system requirements. We also mentioned the wish to link key aspects of system requirements to safety arguments. However, there is a need to be clear about precisely which form of safety arguments are of chief concern. The MOD Defence Standard 00–55 (UK Ministry of Defence, 1995a,b) requires two forms of argument:

(a) deterministic arguments, which are essentially arguments that the Software Requirements Specification (SRS) is logically correct;
(b) arguments based on observing the behaviour of the SRS.

The second form of arguments are essentially derived from testing the implemented system, and are not covered in this paper. The first form of argument are of an analytic nature and are the concern of this paper.

Safety arguments are sophisticated and may combine many different forms of evidence. Certainly, they must include at least some description of the requirements (describing what we would like) and the system itself (describing what the designers intend us to get). This minimal classification is enough to distinguish some key problems in representing safety arguments:

1. How can we be precise in our description of requirements or systems without alienating our clients, who may not be familiar with the formal languages needed for this task?
2. How can we relate requirements to system descriptions, given that the concepts used to describe each may be different?
3. Since safety arguments are seldom watertight, how can we be more explicit about our residual uncertainty and the debate leading up to commitments in our current design?

These are basically a re-expression of the basic problems of requirements engineering (see, e.g., Flynn *et al.*, 1997). The user-developer culture gap is leading to an increasing trend towards providing techniques to enable users to build requirements models (Greenspan, 1995). Yet, especially in safety critical systems, we still need precise descriptions of requirements that can be related to the more formal systems descriptions that the developers need. This leads to problems (1) and (2). Furthermore, there is increasing awareness that requirements are not objective, but emergent (Dobson and Strens, 1994); they are socially constructed by the interactions that take place within the requirements process. Thus we can never be sure that a requirements document is complete and correct in any absolute sense. This is as true of the safety properties of the specified system, as it is of any other aspect. Nevertheless, as experience develops in a certain domain, we may be able to identify certain codes of practice, norms and obligations, which a requirements specification can be checked against. This is essentially the reasoning behind problem (3), and motivates the inclusion of the work that is described in section 5 of this paper. As far as we are aware, these are the key current "meta-concerns" in the requirements engineering domain and so are used as the focus of this paper.

Problems (1) and (3) apply equally to requirements and system descriptions. In this paper, we adopt the simplest approach of using the same formal language to describe both components. This supports a move towards a seamless approach to these two stages in system development. However, it does not entirely solve problem (2) because, although the formal system in which we operate may be uniform, the vocabulary we use for each component may be different.

In the remainder of this paper we examine the extent to which formal languages (of the sort familiar to computational logicians and many knowledge engineers) can help to cope with each of these problems. To make our discussion concrete we give examples of systems which have been targeted at each issue and assess how closely they meet their goals. Although they are valuable contributions, none of these systems give definitive answers to any of the questions we raise. However, they are representative of what can currently be done and give an indication of the limits to what can be achieved with current methods. Section 3 is a response to the first of our problems; it looks at methods for relating natural language to formal language in the context of requirements specifications. Section 4 considers the second problem by demonstrating methods for linking constraints imposed by a code of practice to the elements of construction of a specification for a safety shutdown system. Section 5 addresses our third problem by looking at ways in which we may use formal languages to represent and reason about safety arguments for specifications.

All of the systems we describe are domain specific in the final analysis. In section 3 we are able to provide translations from "natural" to formal language by making two restrictions. Firstly the lexicon is limited to that of a particular domain. Secondly, a fixed association is used between the grammatical structure of English sentences and the syntactic structure of the formal language. If the domain were changed, then it would almost certainly be necessary to alter the lexicon and possibly all the grammatical mappings. In section 4 we connect requirements and systems specifications through structural "schemata" which associate patterns in the system specification with properties which are known to be sought in the requirements specification. Again, if the domain were to be changed, then those properties, and therefore the links, might also change. Even in section 5 the symbols used to describe preferences and obligations would have meanings grounded in particular domains. These would change as we moved between domains.

In consequence, none of the systems presented in this paper provides a definitive, general-purpose answer to the problems we have raised. This is not surprising as in most of the systems the role of domain specific components is to help fit the system to the work practices of a given domain. Although this limits the ease with which the systems can be used without specific domain expertise, it does make them more immediately relevant to application problems. A more extensive discussion of the value of domain-specific formal specification can be found in Robertson (1996).

3 Controlled natural language for requirements specifications

3.1 Features of requirements specifications

Requirements documents for software systems are generally written as natural language documents, perhaps augmented by Data Flow Diagrams (DFD) or some other graphical representation.

Guidelines can be usefully followed to structure the requirements document and ensure that key areas are addressed. In addition, there are some widely used categorisations of the statements that are included in a requirements document. It will be useful to revisit these here for reference during the remainder of the paper.

The key feature of a requirements specification is that it should be primarily concerned with "what" rather than "how". It could be argued that such a document should only concern itself with the function of the intended system. Inclusion of statements of *how* that function be achieved may unreasonably constrain the design process and lead to the development of software which is difficult to port to a different or upgraded environment. Nevertheless, for pragmatic reasons it may be necessary to include such non-functional requirements as part of the requirements document; it may be essential to use a specific processor type, or there may be restrictions on memory usage imposed by cost or space constraints, for example. Indeed, when one is considering safety critical systems high level properties concerning, for example, space, speed, number of active users, robustness and interruptability can become essential requirements and design decisions must necessarily take them into account. We make the following distinction:

- *Functional requirements* are *only* concerned with what the system is to do.
- *Non-functional requirements* are constraints that must be imposed on the design decisions.

Both functional and non-functional requirements must be satisfied by the software design. There may also be additional conditions which are merely used to guide the design process where free choice exists. For example, a preference might be expressed to maximise access time at the expense of increased memory usage, rather than producing a memory efficient piece of code with slower access time. Such requirements are usually referred to as goals:

- *Goals* are conditions which guide the design process where free choice exists.

Just ensuring that the functional requirements, non-functional requirements and goals are clearly identified can add significantly to the effectiveness of a requirements document. However, if these requirements statements are expressed in natural language, they may still be open to problems of ambiguity, incompleteness, imprecision and lack of clarity. There have been suggestions that formal specification languages might be used to eliminate some of these problems. The problem here is that such specifications are not easily communicable to non-mathematicians. Consequently, few domain specialists would be prepared to accept such a document as a contract for the software system (Sommerville, 1992). Fuchs and Schwitter (Fuchs and Schwitter, 1995; Schwitter and Fuchs, 1996) propose the use of controlled natural language to improve the quality of requirements specifications, whilst still maintaining their readability. This has led to the development of the Attempto system.

3.2 The Attempto system

In Attempto, natural language is restricted to a controlled subset with a well-defined syntax and semantics. The intent was to find a subset that was sufficiently expressive for the domain specialists, whilst being accurately and efficiently interpretable by computer. This interpretation follows a two-step process. Firstly a natural language statement is transformed into a *Discourse Representation Structure* (DRS). This enables ambiguities and inter-text references (e.g., anaphora and abbreviations) to be resolved. Secondly, the DRS may be (optionally) translated into a Prolog representation which is executable. By "executable" it is meant that questions may be asked (in controlled natural language) of the specification; the Prolog representation is at too high a level to be considered as a prototype implementation. Although Fuchs and Schwitter do not discuss this explicitly, it would seem that the Prolog representation might also be used as the basis of a detailed formal specification, written initially in Horn clause logic.

The specific natural language used in Attempto is English, so for brevity we will refer to

Controlled English in the remainder of this section. As a minimal model, the following constructs are supported:

- simple declarative sentences of the form subject-verb-object;
- *if-then* sentences;
- *yes/no*-queries and *wh*(*at*)-queries.

In addition, the controlled English text can contain:

- anaphoric references, e.g., pronouns;
- relative clauses, both subject and object modifying;
- comparative clauses;
- compound phrases (and-lists and or-lists);
- negation;
- abbreviations;
- ellipsis as reduction of coordinates.

Attempto has been demonstrated by specifying a simple automated teller machine "SimpleMat" (SM). The resolution of possible ambiguities can be demonstrated with the following fragment of the specification.

```
The customer enters a card and a numeric personal code. If it
is not valid then SM rejects the card.
```

Here the anaphoric reference via the pronoun "it" is most likely intended to be to the personal code, but it could be the validity of the card that is being questioned. Attempto contains a *Parser* and a *Discourse Handler* which together check sentences for syntactical correctness and analyse and resolve inter-text references. A paraphrase explaining how Attempto has interpreted the input is also generated and presented back to the user for checking. In this case, the paraphrase is:

```
the customer enters a card and the customer [same object]
enters [same predicator] a numeric personal_code.

if it [personal_code] is not valid then sm [simplemat] rejects
the card [same object].
```

If this statement does not conform to the user's intent then they must modify the original specification to generate a revised interpretation.

Whilst parsing a sentence, the parser must check for correctness against a predefined Definite Clause Grammar together with a predefined lexicon. The lexicon can be modified via an appropriate editor, which may be called during the parsing process if an undeclared term is encountered. The resolution of references between sentences is effected by using Discourse Representation Theory to provide a semantic structure for a given sentence in the context of the preceding sentences (Kamp and Reyle, 1993; Covington *et al.*, 1988). Our running example yields the following Discourse Representation Structure (DRS):

```
[A, B, C, D]
customer(A)
card(B)
enter(A, B)
numeric(C)
personal_code(C)
enter(A, C)
named(D, simplemat)
IF:
  []
  NOT:
      []
      valid(C)
  THEN:
    []
    reject(D, B)
```

The DRS has the important function of forcing the resolution of any ambiguities present in the Controlled English representation. Attempto has fixed strategies for doing this, and if the user wishes to alter the interpretation they must alter the structure of the input sentence, as indicated above. This is a valuable contribution, but there is another benefit of using this representation. The DRS can be straightforwardly translated into Prolog clauses. In Attempto these are wrapped up in a term `fact/1` and asserted into the knowledge base. In the case of SimpleMat we have:

```
fact(customer(0)).
fact(card(1)).
fact(enter(0, 1)).
fact(numeric(2)).
fact(personal_code(2)).
fact(enter(0,2)).
fact(named(3, simplemat)).
fact((reject(3,1):- neg(valid(2)))).
```

Note that the discourse referents are replaced by Skolem constants (0, 1, 2, etc.).

Once translated into Prolog, the specification can be "executed". That is, the Prolog database can be queried with events being asserted to build up a trace of behaviour. Figure 3 provides a schematic of the overall system.

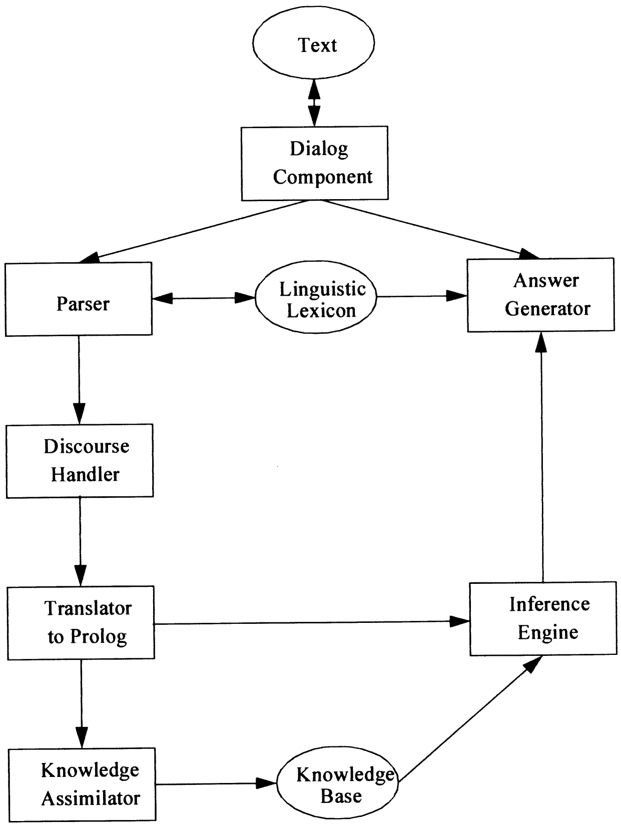

Figure 3 Overview of the Attempto system

3.3 Critique

Attempto seems best fitted to deal with the purely functional component of a requirements specification. Non-functional requirements and goals may, of course, be expressed in Controlled English. However, it seems that Attempto will merely be able to record these statements as assertions without supporting any further analysis of their potential impact on design decisions and possible behaviours of completed systems. As yet there are no published accounts of using Attempto on a significantly sized application. Perhaps the most significant contribution of Attempto is that it enables a requirements specification to be compiled into a logical theory. This may then be queried and reasoned over using the automatic theorem prover Prolog.

This ability to reason over the requirements specification means that some aspects of the validation of the specification can be automated. One could, for example, identify certain general properties that a certain class of applications should satisfy, and check to see if an instance of this class specified using Attempto conformed to these norms. However, as it stands, Attempto offers no specific mechanism to support such a function. In the next section, we shall review one approach to handling this problem.

4 Automated reasoning support for design

4.1 Description of the requirements assistant

The work described in the previous section provides support for eliciting an unambiguous requirements specification, with there being some capability for reasoning about the resulting specification. An obvious next step is to see if some automated support can be given to the generation of a more detailed system design which satisfies the requirements specification. If one is looking to develop safety critical, or other high integrity, systems, then there may also be scope for using some formalisation of regulations or guidelines to constrain further the design choices that may be made. In this section we will look at a Requirements Assistant (RA) which is an interactive system for formalising and managing requirements, including guideline and regulatory requirements (Hesketh *et al.* 1997).

RA is a toolset which assists designers in managing the process of working with large complex collections of requirements, with particular emphasis on safety critical systems. The system has been demonstrated in the context of the development of a specific safety critical application: the design of emergency shutdown systems for drilling rigs. However, the techniques are generally applicable and could be usefully explored in the context of software rather than hardware development.

Given an outline functional specification, relevant requirements from guidelines, or some other corpus of knowledge, are found automatically and checked before being notified to the designer with a note of whether or not they are currently satisfied. As design proceeds, progress in satisfying requirements is monitored automatically and contributing choices are recorded. Such evidence of adherence to guidelines is an endorsement of the validity of the design. In the absence of (or perhaps in addition to) formal guidelines or regulations, one might look to developing a corpus of knowledge based on past experience to provide similar validation checks. During any subsequent system modification, reference to this information can subsequently aid designers by drawing attention to the implications changes will have on maintaining guideline satisfaction.

RA has been demonstrated using guidelines from Shell's code of practice for emergency shutdown systems. We will look at an example where the device being controlled is a turbo-generator, and the sensor that might initiate shutdown of the device is a drilling warning sensor. Some general requirements from Shell's code of practice that might be applicable include:

G1. All trip demands cause shutdown of the appropriate outputs.
G2. 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.

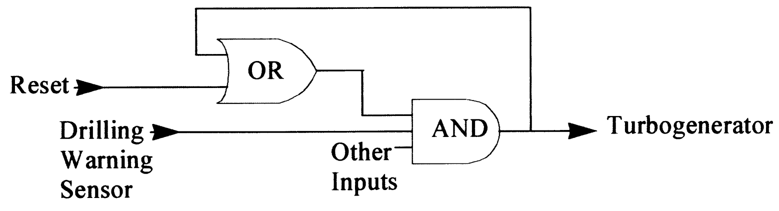

Figure 4 Logic diagram of latched reset

G3. Field devices may be reset. Reset shall be achieved by the provision of local push-buttons incorporated into each output control device.

From G2 a human designer would conclude that the drilling warning sensor should be in logic level 1 during normal operation, but latch to 0 when an error occurs. This is normal fail-safe practice. Furthermore all the signals which could trip the turbo-generator (including that from the drilling warning sensor) should be logically ANDed. This will ensure that any single error will initiate a trip and cause shutdown of the device, thus satisfying G1. To complete satisfaction of G2, the output of the AND gate should be fed back as one of its inputs, thus ensuring that the error condition persists.

G3 now remains to be satisfied. In order to override the error condition, the reset signal needs to be at logic level 1 and must cancel the logical 0 persisting from the error state. This can be achieved by ORing the reset and the latch and feeding this into the AND gate instead of the raw latch signal. Figure 4 illustrates the resulting logic diagram.

The top-level goal of the Requirements Assistant is to provide support for some formalisation of a set of requirements, and then assist in ensuring that any specific design meets those requirements that are relevant to the design context currently under consideration. RA requires that the formalisation of requirements be in a sequent-based logic which is decidable, but other than that imposes no constraints on the formal language chosen. It provides a tool which allows a designer to read an English text on-line, mark phrases or sentences, and then record their translation in whatever formal language has been chosen. Note that this translation is carried out by hand and not mechanically, as in Attempto.

Although RA does not provide a mechanical translation of the requirements, once translated the formalisation is supported by:

- an interface which allows the designer to browse through the satisfied and outstanding requirements;
- links back from requirements expressed formally to the (viewable) natural language from which they originated, thus allowing designers to see the formalised parts in context;
- a theorem prover which can display complete and incomplete proofs, attempt automatic proofs and accept designer instructions;
- display of the formal language either as sequents or in a simple translation to a pseudo-english translation.

The intent is that it be left open to whoever maintains the design support system to use a more detailed formalisation for critical parts of the system and less detailed ones elsewhere. However, experience to date has been focused on the use of a restricted form of first order logic. We will make some suggestions as to alternative logics that might fruitfully be used later on in this paper.

Let us now revisit the running example. It was mentioned above that the formal language used must be sequent-based. In fact, the sequents used are a slight adaptation of the standard interpretation of a sequent calculus in that the hypothesis list is used to specify the context in which a requirement is applicable. In addition, the language used in the demonstrator is augmented with an infix operator "->>" which is used to denote causation. In this language, the requirements G1, G2 and G3 above are formalised as the following four statements:

```
machine(X), initiator(I), expected_consequence(I,X)
⊢ tripped(I) ->> shut_down(X)                                R1.

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

machine(X) ⊢ ∃ r local_reset(r) ∧ resets(r, X)               R3.

machine(X)
shut_down(X) ∧ not_reset(X) ->> latched(X)                   R4.
```

Note that all free variables that occur in the hypothesis list of a sequent are universally quantified across the whole of the sequent. These sequents have a general reading of *Context* ⊢ *Requirement*. That is, if the conjunction of formulae on the l.h.s. of the entailment relation symbol ⊢ is valid, then the requirement on the right hand side applies.

It should be emphasised again that R1–R4 provides *a* formalisation of the requirements G1–G3, but not necessarily the only one. The selection of the context for any requirement is a matter of judgement. If the contexts are too general, the designer will be deluged with irrelevant requirements. On the other hand, if they are too specific they might not ever apply until the requirement was close to being satisfied anyway. In addition, the precise statements that are elicited from a set of requirements is also a matter of judgement. The statement R2, which says that if a machine has been reset and all sensors that control the machine signal an OK signal then the machine will return to operation, could be omitted with the remaining set of statements superficially providing a satisfactory account of G1–G3. In fact, making explicit such implicit requirements can be an important function of the formalisation process. However, precisely what needs explicating, and how, are matters of judgement and these elements of judgement have an important implication. Satisfaction of all relevant sequents should not be regarded as *proof* that a design satisfies the corresponding English language code of practice. Rather, they are arguments supporting the validity of a design. With the relatively lightweight formalisations used in RA the usability of the system is increased, but the full precision of detailed formalisation is sacrificed.

At this stage, we have the machinery to check a design to see if all relevant requirements are satisfied, and to prompt the designer with those that are not. However, before this checking can be carried out, we must be able to specify the components in a design. The RA contains files of schemata that specify the objects that can be selected and added to a design. In fact, as we shall see later, RA itself can select objects to propose as solutions to unsatisfied requirements.

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. These contributions are phrased in the same language as the requirements, and the objects are specifically designed to provide contributions which explicitly meet one or more of the formal requirement statements. An example of a schema which is relevant to our running example is that of a latched reset circuit 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. This particular schema demonstrates the possibility of parameterisation with the variable Is corresponding to a list which could be instantiated to any number of initiators. It corresponds to the circuit shown in figure 4 where the initiators Is are ANDed with the signal that results from the ORing of the local reset and the AND gate output.

name:	latchedreset
context:	machine(O)
	Is = {X | initiator(X) ∧ expected_consequence(X,O)}
	reset(R)

```
               local_reset(R)
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) ∧ ∀ j : (on(j, Is) ∧ ok_signal(j)) - > > ok_signal(O)
               [] resets(R, O)
```

Note that [R, O] denotes the list containing only the two elements R and O, whilst [Or| Is] denotes the list constructed from adjoining the element Or to the beginning of the list Is.

The latchedreset schema has four explicit contributions. However, these can provide more than four contributed facts when instantiated if there are multiple ways of satisfying the hypotheses derived from the Code of Practice (R1, . . . , R4 in this case).

Note also that of there is more than one intitiator in the Is list, RA will make an instantiated copy of the first contribution for each one. It is now straightforward to see that incorporation of an appropriate instance of this schema into a design for a turbo-generator control circuit will ensure that the formal statements R1–R4 are satisfied.

Using the above basic machinery, RA can provide support for the following management tasks.

Noticing when requirements apply
As objects are added and design steps taken, RA checks for relevant requirements incrementally.

Keeping track of all the different requirements
The RA maintains lists of all the satisfied and the unsatisfied requirements being invoked. When RA has established that a requirement is satisfied, it is added to the list of satisfied requirements together with the argument for its satisfaction. If RA concludes that a requirement is not satisfied, the instance and any partial argument relating to its satisfaction are put on a list of outstanding requirement obligations. When a new design step is taken, not only must any new requirements be recorded and checked, but the implications for existing satisfied and unsatisfied requirements and their associated arguments must also be evaluated. This automated book keeping is an important contribution of RA.

Assessing the extent to which requirements are currently satisfied
A designer may examine complete as well as partial arguments for a requirement's satisfaction. The designer may examine and work on these proofs interactively.

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 RA's database, RA can offer advice by second guessing. The advice is calculated by looking at the applicable schemas' contributions and seeing which schema satisfies the greatest number of outstanding requirements in relation to the current context. If there is a tie, it is resolved by preferring solutions which do not add unexpected contributions, since they are likely to add unnecessary complexity.

Evidence of requirements' satisfaction
The evidence is the proof, or argument, for each hypothesis derived from the Code of Practice which RA has automatically constructed using the facts supplied by the designer's addition of objects and their contributions and relationships. Since positive evidence is needed to back up averrals of requirements' satisfaction, only constructive proofs are admitted. That is, proofs which only infer from things which exist or which can be constructed from things which exist.

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 contributions of any component can be inferred by looking at all the requirement arguments in which it occurs.

Remembering requirements' influence on the design decisions
As well as knowing the 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 the arguments are dependent on the components which appear in them.

4.2 Critique

The formalisations of requirements in RA have no guarantees that they are strict translations of their corresponding english language statements. Links are present in RA relating formalisations to their associated text statement in the Code of Practice so that a designer can check their decisions against this. It may be worth exploring the possibility of integrating RA with a controlled natural language front end to support requirements elicitation and expression in a more readily formalisable form, although such a system will make great demands on the language support.

The RA has been demonstrated in the context of the development of control circuitry for safety shutdown systems. The systems of concern are hardware, but the potential for applying RA to software development is a specific interest in the context of this paper. This has not been demonstrated conclusively but one can envisage producing a corpus of general, formalisable, software requirements that are applicable in specific contexts. The relevant schemata could then be formalisable program fragments, skeletons that may be refined into specific instances which provide specified contributions to a software design. In addition, the basic representational framework of RA is a development of the technology used in the EcoLogic (EL) Project (Robertson *et al.*, 1991). EL used schematic definitions of software modules to enable programs for running ecological models to be interactively generated. System development is controlled differently in RA and EL. Nevertheless, EL provides a strong witness to the validity of the assertion that RA could be used for software development.

Referring back to the discussion of validation in the critique of Attempto in section 3.1, one of the key features of RA is that it does provide some support for the validation of a design. Because RA does not have a rigorously defined mapping between the english language code of practice and the formal representation, it cannot guarantee validity. However, it can assist in identifying some of the main points at which an auditor might wish to validate a designer's understanding of some part of the code of practice, or to verify that a particular component possesses a behaviour consistent with key parts of the code of practice. This is possible because RA maintains links between schemata of the design and appropriate formal interpretations of the code of practice. It is useful because one of the difficulties in the validation of such systems is simply remembering that the designers have paid attention, at the right points, to the critical guidelines.

5 Modal languages for automating safety reasoning

5.1 Modalities for Requirements Analysis

We will explore some ideas on using modal logics to extend the language for reasoning about safety in this section. There are a number of reasons for wishing to use modal logics. Firstly, for example, we might wish to allow a requirements engineer to distinguish at the language level between statements of different types; in particular, functional requirements, non functional requirements (obligations) and goals (recommendations or preferences). Secondly, it is quite often necessary to incorporate some temporal reasoning in requirement statements. We will illustrate this latter with a simple example. This example is motivated by a real case which had tragic consequences. In the real case the complexity of the conditions leading to failure were much more subtle than the following (Leveson, 1995). The simplification is for ease of exposition, but the original example is real.

A medical radiotherapy system used a high energy radiation source. This could be run in two modes. In what we will call mode1, the source was run at low power, with the beam targeted directly at the patient. In mode 2, a lead shield was placed between patient and source with the latter being run at high power to stimulate secondary emission from the screen. Both source and shield were controlled by software. Unfortunately, it turned out that with a rarely occurring sequence of commands, it was possible for the source to be run at high power instantaneously before the shield was in place. Although the steady state conditions were repeatedly checked, it was this transitory condition which resulted in several patients receiving severe, and in some cases fatal, radiation burns.

We can provide an initial simple formalisation of the situation as follows:

```
mode1 ← shield_open ∧ low_power
mode2 ← shield_closed ∧ high_power
shield_open ∨ shield_closed
low_power ∨ high_power
```

As it stands, `shield_open ∧ high_power` is consistent with this theory. We can further refine the theory by adding the constraint:

```
¬(mode1 ∧ mode2)
```

This will rule out the problematic state as a steady state condition. However, we are not placing any ordering on the events which may lead to valid steady states, and a valid[4] implementation may still produce transitory states that are potentially lethal. What is needed is a further statement checking that the shield is always in place *before* the source is switched to high power.

5.2 $\mathbf{L}_{safe}$

A number of approaches have been taken to the development of implementable temporal logics (see Fisher (1996) for a review). That described in Das and Fox (1993) and Hammond *et al.* (1994) is of particular interest in the present context as it contains a number of modalities specifically targeted at reasoning about safety. We will briefly review it here.

Safety reasoning will in general involve a set of temporally qualified assertions outlining both what is known about a universe of discourse in different periods of time, and the consequences of actions in different conditions. $\mathbf{L}_{safe}$ was developed to enable such assertions to be expressed as a logical theory. In order to describe both *static* and *dynamic* aspects of the universe of discourse, the propositional symbols in $\mathbf{L}_{safe}$ are sorted into properties and actions. In addition, the language of propositional logic is extended with a number of modalities, including a temporal operator of the form $[t_1, t_2]$, where $t_1 \leq t_2$ and t_1, t_2 are non-negative integers. The full set of modalities, with informal readings, is as follows:

[RECO]α	Action α can be recommended
[SAFE]α	Action α is safe
[AUTH]α	Action α is authorised
[PREF](α,β)	Action α is preferred to action β
[OBLG]φ	Action φ or property φ is obligatory

[4]"Valid" in the sense that the implementation is a model of the theory.

$[t_1, t_2]\varphi$ Action φ is taken, or property φ is true in the interval t_1 to t_2.

Some additional notation is provided for brevity of expression. If α is an action and $[t_1, t_2]$ is the smallest interval over which that action is taken, then $[t_1, t_2]\alpha$ may be written as $[t_1, t_2]!\alpha$. In addition, an interval of the form [t, t] represents a time point, and may be written as (t).

We may now write a more satisfactory constraint for our requirements theory concerning the hypothetical medical scanner.

```
∀ t₁,t₂([t₁, t₂]high_power →
∃ t₃,t₄(t₄<t₁ ∧ [t₃,t₄]close_shield ∧ [t₄,t₂]shield_closed))
```

This says that the `close_shield` action must have been carried out before a `high_power` state is assumed, and that the shutter must remain closed at least for the period during which the `high_power` state holds.

The modality [OBLG] is taken from Deontic logic (Chellas, 1980). This is typically used to describe norms of behaviour; what one *ought* to do. It can be thought of as a "strong" modality whose behaviour is to a certain extent analogous to the necessity modality $\Box$, or universal quantification, $\forall$. However, as used in $\mathbf{L}_{safe}$, there is an important distinction which is of interest in the present context. In classical modal logic, by saying a sentence φ "necessarily holds" it is meant that φ holds in every situation including the present one. This is characterised by the axiom $\Box\varphi \rightarrow \varphi$. However, $[\text{OBLG}]\varphi \rightarrow \varphi$ is *not* an axiom in $\mathbf{L}_{safe}$. This has a rather important ramification; a theory will still remain consistent even if both $[\text{OBLG}]\varphi$ and $\neg\varphi$ are derivable. Rather than this being regarded as implying a contradiction, this is regarded as a "hazardous state" of the knowledge base, represented by the special symbol $\otimes$. That is, if $[\text{OBLG}]\varphi$ is derivable from a knowledge base, then non performance of φ, if φ is an action, or non-satisfiability of φ, if φ is a property, leads to a hazardous state. This is expressed formally by the equivalence:

$$[\text{OBLG}]\varphi \equiv \neg\,\varphi \rightarrow \otimes$$

The usage of the [OBLG] modality in constraints can be further clarified by considering a notion of "technological" possible worlds (Das, personal communication). A conventional propositional statement can be used as a hard constraint in the possible worlds that satisfy a certain logical theory; we could fairly insist that a person's height have only positive values, for example. In contrast, although we might normally require that a person be within a certain age range before being admitted into a cancer therapy trial, this is a *technological* constraint, and not an inherent property of the world. As such, a particular patient might violate the constraint, having been admitted onto the trial for certain pragmatic or judgemental reasons. Although this world is technologically unsafe, it is not impossible. This latter form of constraint may be distinguished by the use of the [OBLG] modality in $\mathbf{L}_{safe}$.

This has potential value in the formalisation of requirements. We may use the [OBLG] modality to distinguish non-functional from functional requirements. A valid design must necessarily satisfy the functional requirements in a specification. However, if the non-functional requirements are expressed as obligations, this leaves scope for a valid but unsafe design in which one or more of the non-functional requirements is not satisfied. In particular, we may have a rapid prototype which demonstrably satisfies the functional requirements, whilst satisfaction of time and space constraints is temporarily suspended.

The modalities [RECO] and [PREF] lend themselves to the expression of goals. In $\mathbf{L}_{safe}$, a recommended action is not obligatory. $[\text{RECO}]\alpha$ merely expresses a recommendation that α should be made to come about, if it is a property, or occur, if it is an action. The user is free to choose whether or not to act upon this recommendation. In a similar way, $[\text{PREF}](\alpha, \beta)$ merely expresses a preference of α over β, without any enforcement of this preference. Thus we might use

[PREF]`(low_memory_requirement, high_memory_requirement)`

to express the goal to prefer design solutions which minimise memory requirements, given that there are no overriding constraints. Similarly, we might use [RECO]`high_reliability` to express the goal that a system should be as reliable as possible.

5.3 Critique

$\mathbf{L}_{safe}$ is one of a number of temporal modal logics that have been proposed for use in describing high level timing requirements. One can view a set of timing constraints expressed in a language such as $\mathbf{L}_{safe}$ as expressing a logical theory of the application being developed. A detailed specification in, say, Timed CCS (TCCS) (Wang Yi, 1991) is then a (possible) model of the logical theory. One then has a formal framework for verifying that the specification is indeed a model of the theory. In the case of timed process algebras such as TCCS, a standard approach to this is to run a simulation of the specification, and see if all possible execution traces satisfy the formulae of the logical theory. If it is established as a valid model, one may then also exercise the model further to see if it has any additional, but undesirable, properties; hence using it as a validation check on the requirements specification.

There is a considerable literature on these approaches to the specification and design of real time systems (although less in the way of large scale practical experience). See Ostroff (1992) for a fairly recent review. The difficulty is that the rather brute force approach of model checking limits the size of specification that can be handled in this way (e.g. Larsen *et al.*, 1995), although recent experience indicates that many problems can be decomposed into manageable modules (Roscoe, 1994).

We have chosen to focus on $\mathbf{L}_{safe}$ because the temporal logic is embedded in a much more expressive language than most (Modal Action Logic, developed as part of the FOREST project (Atkinson *et al.*, 1991) also includes the use of deontic modalities). These additional modalities seem to have a natural application to requirements specification, although experience with their use does not at present extend outside the domain of clinical protocol specification. We would like to see this changed.

6 Conclusions

This paper has two purposes. The first is to illustrate a number of styles of approach that may be taken to pushing a higher degree of formality back towards the earlier stages in the software development process. The second is to use these examples to illustrate three key problems in representing safety arguments:

- how can we resolve the tension of developing a language that is specific enough for the domain experts to be comfortable with, yet which is formal enough to admit detailed analysis;
- how can we relate requirements to system descriptions, when the concepts used to express each may differ;
- how can we make explicit the uncertainty that necessarily remains, given that safety arguments are seldom watertight?

We have focused on using formality in analysing and criticising requirements with regard to safety properties and safety arguments. But the discussion can be equally applicable to any mission critical aspects of a complex system.

As mentioned at the outset, none of the solutions we offer is a complete one. Although they each address one of the above questions, none of them attempts a general solution. In addition, they all ultimately have a meaning grounded in the intended domain of application. One should not be too pessimistic about this last, however. It may well be that a definitive solution is not necessary. Rather, it is a matter of striking an appropriate balance, based on the expertise of the domain experts and the development team. It is perhaps more important to focus on producing a unification of the positive aspects of the individual results discussed in this paper.

A further refinement would be to extend the expressiveness of the uncertainty handling. The proposal of this paper (section 5) is to allow uncertainty to be expressed by weakening categorical logical arguments by allowing certain obligations, recommendations and preferences to be expressed. The mere fact of making the arguments explicit and open for inspection, of course, also means one allows the possibility that someone may doubt the validity of a claim warranted by a certain line of reasoning. Actually giving some *measure* of residual uncertainty is, however, a much more contentious issue, given that one is often predicting on the basis of analogy or sparse data. However this is an aspect which needs to be explored further.

All of the systems in this paper demonstrate that it is possible to find niches for formal methods in the early stages of software lifecycles and that this need not involve revolutionary change to existing practices. Whether this possibility becomes an actuality depends on our ingenuity in embedding them within an appropriate safety culture. Other papers in this edition touch on this wider issue.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

Atkinson, WD, Booth, JP and Quirk, WJ, 1991, "Modal action logic for the specification and validation of safety" in de Neuman, B, Simpson, D and Slater, G (eds) *Mathematical Structures for Software Engineering* Oxford: Clarendon Press.

Blackburn, JD, Scudder, GD and Van Wassenhove, LN, 1996, "Improving speed and productivity of software developers" *Proc. IEEE Trans. Software Engineering* **22** 875–885.

Chellas, B, 1980, *Modal Logic* Cambridge University Press.

Covington, MA, Nute, D, Schmitz, N and Goodman, D, 1988, "From English to Prolog via Discourse Representation Theory" *Research Report AI-1994–06*, Artificial Intelligence Centre, University of Georgia.

Das, SK and Fox, J, 1993, "A logic for reasoning about safety in decision support systems" *Proc. European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU '93).*

Dobson, J and Strens, R, 1994, "Organisational requirements definition for information technology systems" *Proc. First IEEE International Symposium on Requirements Engineering (ICRE '94)* Colorado Springs, CO, 158–165.

Fisher, M, 1996, "An introduction to executable temporal logics" *The Knowledge Engineering Review* **11** 43–56.

Flynn, DJ, Jazi, MD and Quek, P, 1997, "User evaluation of a user-centred modelling method by case study" *Proc. EASE-97: Empirical Assessment in Software Engineering* University of Keele, England.

Fox, J, 1993, "Engineering safety into expert systems" in F. Redmill and T. Anderson, (eds.) *Safety Critical Systems* Chapman and Hall.

Fuchs, NE and Schwitter, R, 1995, "Attempto: controlled natural language for requirements specifications" *Proc. Seventh ILPS Workshop on Logic Programming Environments* Portland, Oregon.

Gough, PA, Fodemski, SA, Higgins, SA and Ray, SJ, 1995, "Scenario—an industrial case study and hypermedia enhancements" *Proc. Second IEEE International Symposium on Requirements Engineering (RE '95)* York, England, 10–17.

Greenspan, S, 1995, "The next 25 years: new customers, new environments, new requirements" *Proc. Second IEEE International Symposium on Requirements Engineering (RE '95)* York, England, 36–37.

Hammond, P, Harris, AL, Das, SK and Wyatt, JC, 1994, "Safety and decision support in oncology" *Meth. Inf. Med.* **33**(4) 371–387.

Hesketh, J, Robertson, D, Fuchs, N and Bundy, A, 1997, "Automating reasoning support for design" *Automated Software Engineering* (to appear).

Kamp, H and Reyle, U, 1993, *From Discourse to Logic: Introduction to Model Theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory* Dordrecht: Kluwer Academic.

Larsen, KG, Petterson, P and Yi, W, 1995, "Compositional model checking and real-time systems" *Proc. 16th Real Time Systems Symposium* Pisa, Italy 5–7 December.

Leveson, N, 1995, *Safeware: System Safety and Computers* 515–553. Addison-Wesley.

Maisey, D and Dick, J, 1996, "Measuring the quality of the development life cycle process" *Software Quality Journal* **5** 199–210.

Ostroff, JS, 1992, "Formal methods for the specification and design of real-time safety critical systems" *Journal of Systems and Software* **18**(1) 33–60.

Robertson, D, Bundy, A, Muetzelfeldt, R, Haggith, M and Uschold, M, 1991, *Eco-Logic: Logic-Based Approaches to Ecological Modelling* MIT Press.

Robertson, D, 1996, “Domain specific problem description” *Proc. 8th International Conference on Software Engineering and Knowledge Engineering* Nevada, USA.

Roscoe, AW, 1994, “Model-checking CSP” in Roscoe, AW (ed.) *A classical mind: essays in honour of C.A.R. Hoare* Prentice Hall, 353–378.

Schwitter, R and Fuchs, NE, 1996, “Attempto: from specifications in controlled natural language towards executable specifications” *EMISA Workshop, Natürlichsprachlicher Entwurf von Informationssystemen* 28–30 May, Ev. Akademie Tutzing.

Sommerville, I, 1992, *Software Engineering. Fourth Edition* Wokingham: Addison-Wesley.

UK Ministry of Defence, 1995a, “The procurement of safety related software in defence equipment” *Technical Report DEF STAN 00–55* (Part 1: Requirements).

UK Ministry of Defence, 1995b, “The procurement of safety related software in defence equipment” *Technical Report DEF STAN 00–55* (Part 2: Guidance).

Wang, Yi, 1991, “CCS + time = an interleaving model for real time systems” *Proc. International Colloquium on Automata, Languages and Programming (ICALP ‘91)* Madrid, Spain.

Woodcock, J and Davis, J, 1996 *Using Z—Specification, Refinement and Proof* Hemel Hempstead, UK: Prentice Hall.