Literate Specification: Using Design Rationale To Support Formal Methods In The Development Of Human-Machine Interfaces*

C.W. Johnson
Department of Computing Science, University of Glasgow,
Glasgow, United Kingdom, G12 8QQ.
E-mail: johnson@dcs.glasgow.ac.uk

ABSTRACT

The design of safety-critical user interfaces is, typically, very different to that of many other applications. Reactor control systems and aircraft cockpits are complex and dynamic; they are open to input from many different users and devices. A number of formal notations, including Z and temporal logic, have been developed to address these problems. They provide precise and concise means of representing a potential design before designers incur the expense of implementation. In consequence, government bodies and commercial organisations have recommended that these techniques be used when tendering for their contracts. There are, however, a number of limitations that restrict the use of mathematical specifications for interface development in large scale projects. In particular, formal notations cannot easily be used to coordinate the activities of human factors and systems engineering teams. This creates particular difficulties if some group members only have a limited understanding of discrete mathematics. A further problem is that the development of a safety-critical applications may take many months, or even years, to complete. This creates difficulties because abstract, mathematical specifications cannot easily be used by new members of a development team to understand past design decisions. In order to avoid these limitations we have developed a 'literate approach' to interface specification. This technique uses a formal development language and a semi-formal design rationale to support the design of safety-critical user interfaces.

Keywords: *formal methods; design rationale; human computer interaction; safety-critical systems.*

1 Introduction

The development of safety-critical systems is qualitatively different from that of many other interactive applications. For instance, aircraft cockpits and reactor control rooms are more complex and dynamic than single-user text-editors or spreadsheets [29, 31]. These differences impose considerable burdens upon the design teams that must ensure the safe and successful development of human-machine interfaces.

1.1 Formal Specifications

Formal methods have been developed to support the design of safety-critical systems [2]. They can be used to strip out low-level implementation details, such as device handling, during the early stages of development. This enables designers to focus in upon more 'significant' properties of a potential implementation [8]. In consequence, formal methods are increasingly being used in the development of large scale applications. For instance, NASA are applying mathematical specification techniques to represent and reason about a range of on-board communications systems [28]. Formal analysis is a necessary part of many design projects commissioned by the UK Ministry of Defence [25]. Formal methods have also been used to support the development of interactive systems. For instance, Sufrin and He have extended the Z schema calculus to represent interaction with a text-editor [34]. Took has

*Submitted to the Human Computer Interaction journal.

developed a precise formal framework for the representation of graphical interfaces [36]. A number of authors have extended these techniques to support the development of safety-critical, user interfaces. For instance, Fields, Wright and Harrison have used CSP to identify human-error tolerance requirements for an aircraft fire-fighting system [9]. Johnson, McCarthy and Wright and have exploited formal methods to identify human factors problems that were neglected in the Kegworth accident report [20]. Johnson has also developed a number of notations that are specifically designed to capture temporal and probabilistic properties of interaction with safety-critical systems [13, 14, 19].

A number of limitations restrict the application of formal specification techniques to support the development of large scale, safety-critical user interfaces. Some of these limitations arise from the technical problems of constructing complex, mathematical specifications. These issues are dealt with elsewhere [1, 27, 17] and are briefly described in Section 3 of this paper. Other problems stem from the difficulty of using abstract, mathematical notations to coordinate the activities of different development teams. This is a significant issue because team-based management structures are, typically, used to organise the development of safety-critical applications [21]. These groups can be organised around specific sub-systems. Alternatively, they can be formed around different skill-bases; human factors groups must coordinate with systems engineering teams. In either case, many of the people in these groups will have little knowledge of discrete mathematics. This prevents them from understanding the implications that a formal specification may have for their development tasks. The use of a formal notation, therefore, jeopardises communication both within and between design teams [16]. A further problem is that while formal specification techniques provide powerful means of representing interactive behaviour, they cannot easily be used to describe **why** that behaviour is appropriate. This creates problems for designers and for the regulatory authorities that must understand the motivation behind particular development decisions.

1.2 Design Rationale

A number of semi-formal notations have been proposed to explicitly represent the justifications that lie behind behind particular development decisions. For instance, the graphical Issue Based Information System (gIBIS) uses a complex web of issues, positions and arguments to record the rhetorical events in a design meeting [5]. In contrast, Figure 1 illustrates the simpler syntax of the Questions, Options and Criteria (QOC) notation [32]. These diagrams are built by identifying the key questions that must

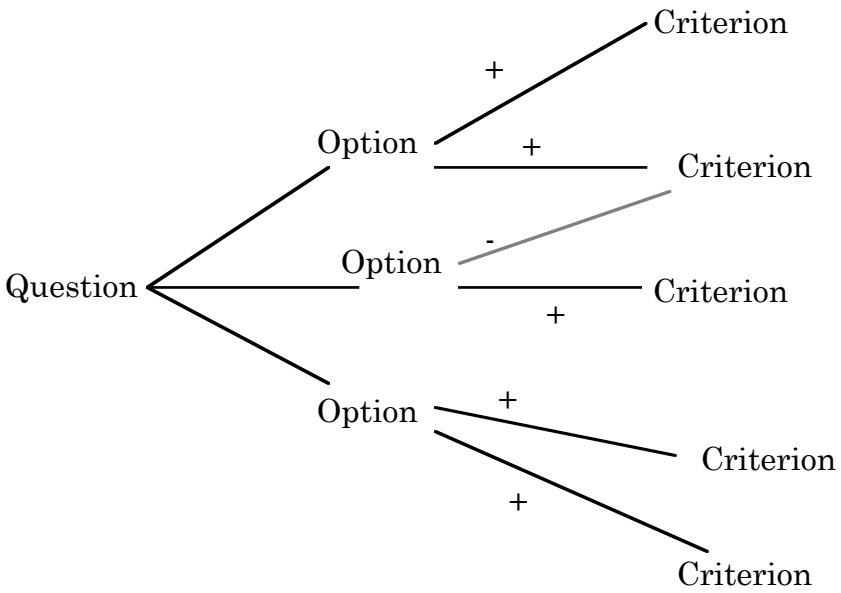

Figure 1: A Simple QOC diagram.

be addressed during the development of an interactive system. The options that answer a particular question are then linked to it using the lines shown in Figure 1. Finally, options are linked to the criteria that support or weaken them using solid or broken lines.

Design rationale notations offer a number of advantages for the development of safety-critical user interfaces. Natural language is used to label the nodes in both QOC and gIBIS. This provides the

flexibility that is necessary in order to represent the vast range of decisions that have to made during the development of large-scale systems [32]. A further advantage is that designers are not forced to learn the complex syntax and semantics of mathematical specification languages. This helps to avoid the communications problems that restrict the application of formal notations, such as Z and temporal logic [16]. MacLean, Young, Bellotti and Moran regard the arguments that support development decisions as the 'co-products of design' [23]. If the justifications that support crucial decisions are explicitly represented then they can be shown to new members of development teams [5]. Yakemovic and Conklin argue that this helps to clarify the reasons for past development decisions [38]. Design rationale can also be shown to regulatory authorities as part of the safety case for an interactive system [7]. This helps to demonstrate that sufficient attention has been paid to the development of a human-machine interface.

Unfortunately, design rationale notations lack the precision that is required during the development of safety-critical systems. The natural language that is used to annotate QOC diagrams can easily be misinterpreted [20]. These notations can also be inconsistent if the same criterion is used both to support and weaken an option in different contexts. A companion paper addresses these problems in more detail [15]. This paper, however, focuses upon the use of design rationale to avoid the weaknesses of formal methods. We introduce a 'literate' approach to interface specification. This extends the literate programming techniques that were first proposed by Knuth [22] and later developed by Thimbleby [35]. Knuth and Thimbleby have developed tools and techniques that explicitly link natural language documentation to the code of computer programs. In contrast, our approach links semi-formal design rationale to the abstract clauses of mathematical specifications. The natural language argumentation in the design rationale provides a means of communicating the findings of formal analyses to the members of concurrent design teams without sacrificing the precision provided by formal development techniques. This approach also helps new members of design teams. The explicit representation of the reasons for and against formal design options can help designers to understand their context of development. It can also be used to help external agencies and quality control groups to monitor progress towards an implementation.

1.3 Outline Of The Paper

Section 1 has introduced the argument that is presented in this paper. Section 2 describes an example application that is used to illustrate the remainder of this paper. The description is kept to a minimum as our focus here is upon the general problems of applying formal specifications to the development of large scale user interfaces. Section 3 briefly describes the benefits that mathematical specifications provide for the development of safety-critical interfaces. Section 4 then identifies a range of management problems that restrict the use of formal methods to support large-scale project management. In particular, it is argued that the lack of vernacular descriptions prevents many designers from accessing the products of formal analyses. A further limitation is that mathematical specifications do not represent the justifications which support particular requirements. Abstract requirements provide little information about the status of a specification; whether it needs further refinement or is sufficient for transformation into an eventual implementation. Finally, it is argued that formal methods fail to represent the alternative options that were considered but rejected during earlier stages of design. This is important if options have to be reconsidered in the light of changes to user or customer requirements. Section 5 takes each of these problems in turn and demonstrates that design rationale can be used to avoid them. The QOC notation is used together with a logic formalism to represent both **what** an interface must do and **why** it must do it. Section 6 describes two different toolsets that are being developed to support our 'literate' specification techniques. Section 7 presents the conclusions that can be drawn from this work and suggests directions for future research.

2 An Example Application

The remainder of this paper uses requirements for a chlorine recovery plant to illustrate the application of risk assessment techniques. This example has been chosen because it typifies the complex and stochastic application processes that pose a considerable challenge to interface designers [37]. In particular, we are concerned to represent and reason about operator intervention to resolve periodic failures in the the pumping equipment. Faults can occasionally develop in the compressors and so maintenance is required in order to guarantee the integrity of the system. If repair activities are prolonged then operators are

required to halt the recovery process. Rather than develop a prolonged description of this application, the interested reader is directed to [7]. In contrast, the remainder of this paper explores the strengths and weaknesses of formal methods for the design of user interfaces to such safety-critical applications.

3 The Benefits Of Formal Methods

Logic notations offer a number of benefits for the development of safety-critical interfaces. There is a close correspondence between specifications written in first order logic and programs that satisfy those specifications written in PROLOG [4]. This provides a link between formal analysis and rapid prototyping. A further benefit is that logic abstractions can be used to strip out the mass of implementation details that obscure important properties during the early stages of design. These details can gradually be introduced as development progresses [12]. For instance, the following clause shows how simple input and output requirements can be represented. Designers can use logic clauses, such as *display* and *select*, to focus upon the structure of interactive dialogues without having to consider the low level device handling that must be supported by an eventual implementation. The following clause states that a failure in a compressor is acknowledged if the compressor has failed and a warning icon is displayed for that component and the user selects that icon:

$$user_select_acknowledge(compressor_failure) \Leftarrow state(compressor, failed) \wedge display(compressor_error_icon) \wedge select(compressor_error_icon) \quad (1)$$

Unfortunately, a number of limitations restrict the utility of first order logic as a means of developing safety-critical, user interfaces. There are a number of technical problems. For instance, there is no notion of sequence in first order logic. The previous clause would be true if the warning icon were displayed before the error had occurred. This is clearly not what the designer had intended. This notational problem can be avoided through the use of temporal logic operators: $\bigcirc$ (read as 'next'); $\Diamond$ (read as 'eventually'); $\Box$ (read as 'always'); $\mathcal{U}$ (read as 'until'). The following clause illustrates how these operators introduce the sequencing that is missing in first order logic. It is important to emphasise that this use of a formal notation supports communication between designers who have a clear understanding of the logic. Temporal operators, such as $\bigcirc$ and $\Diamond$, possess a clear semantics and can be used to avoid ambiguous, everyday, terms such as 'later' or 'during'. The following clause states that a failure in a compressor is acknowledged if in the present interval the compressor has failed and in the next interval a warning icon is displayed for that component and eventually the user selects that icon:

$$\begin{aligned} user_select_acknowledge(compressor_failure) \Leftarrow \\ state(compressor, failed) \wedge \\ \bigcirc(display(compressor_error_icon) \wedge \\ \Diamond(select(compressor_error_icon)) \end{aligned} \quad (2)$$

Brevity prevents a more detailed introduction to temporal logic; the interested reader is directed to [12, 18, 24]. Informal readings will be provided throughout this paper. In contrast, this paper uses the notation to illustrate the weaknesses of formal methods as means of communicating development decisions amongst the members of concurrent design teams.

4 The Weaknesses Of Formal Methods

There have been few attempts to provide methodological support for the application of formal methods to support interface development [6]. In consequence, there are a number of important barriers that prevent them from being easily applied within large scale development projects.

4.1 Lack Of Vernacular Descriptions

The previous clauses illustrate many of the problems that limit the application of formal methods to support the large-scale development of safety-critical user interfaces. In particular, it is difficult for non-formalists to understand the implications that abstract clauses, such as (2), have for their development tasks. A number of techniques, such as Z and VDM, attempt to address this problem by stipulating that

informal descriptions are a necessary part of any specification [33]. Unfortunately, this requirement is often ignored [16]. One means of ensuring that vernacular descriptions are included within formal specifications is to introduce prose into the syntactic structures that are used to organise abstract requirements. For example, the previous clause might be grouped with an informal description:

$$\begin{aligned} &user_select_acknowledge(compressor_failure) \Leftarrow \\ &\quad state(compressor, failed) \wedge \\ &\quad \bigcirc(display(compressor_error_icon) \wedge \\ &\quad \Diamond(select(compressor_error_icon)) \end{aligned} \tag{3}$$

This states that a failure in a compressor is acknowledged if in the present interval the compressor has failed and in the next interval a warning icon is displayed for that component and eventually the user selects that icon.

Although this syntactic grouping can help to communicate the requirements that are represented in a formal specification, it does not provide a panacea for the use of formal methods within large development teams. It states **what** an interface must do, it does not explain **why** it must do it. In other words, this approach lacks any justification for the design requirement.

4.2 Lack Of Justification

Formal specifications are used to describe interface behaviour; they are not used to 'justify' that behaviour. For instance, the previous clause (3) specified that operators must explicitly select the *compressor_error_icon* in order to acknowledge the warning. This might be justified in terms of the complex working environments that frequently frustrate the operation of safety-critical interfaces [30]. Distractions caused by other operators and application processes can prevent operators from immediately observing error icons. Designers might, therefore, introduce the selection requirement in order to increase their confidence that users will notice the failure. Unfortunately, there is no means of reconstructing this informal argument from the formal statement of the design requirement. This threatens the successful development of complex user interfaces within large development teams. For instance, an alternative technique might be used to automatically remove less significant icons, such as blow back warnings, once an error has been resolved. Blow-back valves prevent substances being forced up a line that they have just been pumped down. Considerable confusion is likely to arise if designers do not understand why manual confirmation was appropriate for compressor failures but not for blow back errors:

$$\begin{aligned} &automatically_remove_warning(blow_back_error) \Leftarrow \\ &\quad state(blow_back, failed) \wedge \\ &\quad \bigcirc(display(blow_back_error_icon) \mathcal{U} \\ &\quad select(blow_back_error_icon) \end{aligned} \tag{4}$$

This clause states that a failure in a blow-back valve is acknowledged if in the present interval the valve has failed and in the next interval a warning icon is displayed for that component until the valve is working again.

This lack of justification has important implications for the application of formal methods. Other designers within the same group or within different development teams cannot use clauses, such as (3) or (4), to discover why one approach was appropriate for compressor failures whilst another was appropriate for blow-back errors. This makes it difficult for the same criteria to be applied throughout the design of a user interface. At worst, this may lead to a series of ad hoc decisions that fall far short of the rigorous techniques advocated by the proponents of formal methods.

4.3 Lack Of Status Information

A further limitation of existing specification techniques is that they provide no means of determining the status of a requirement from its formal statement. By status information, we mean some indication that

a requirement requires further work before it can be used to guide the implementation of a user interface. A prime objective in the use of formal methods is to minimise the modifications that are necessary once a specification has been refined towards implementation. If some requirements are overlooked then there may be considerable delays and additional costs whilst that area of a design is developed to the required level of detail. For example, clause (3) used the $\Diamond$ (read as 'eventually') operator to represent the delay that occurs between the warning icon being presented and the user selecting it. Experience in the use of temporal logic quickly teaches that the $\Diamond$ operator is too vague to support the implementation of safety-critical systems. The previous clause would be satisfied if the selection occurred ten seconds after the warning, twenty minutes after the warning, two days after the warning, etc. In particular, problems might occur if the compressor re-started before the operator observed the error icon. They might then assume that the compressor was broken when it was, in fact, working again. The $\mathcal{U}$ operator can be used to extend the automatic cancellation of clause (4) to the compressor warning. The success of this approach relies upon designers identifying the need to refine the previous version of *user_select_acknowledge*. Unfortunately, there is no way for designers to detect that this additional work is required from the formal statement of (3):

$$\begin{aligned} &user_select_persistant_warning(compressor_failure) \Leftarrow \\ &\quad state(compressor, failed) \wedge \\ &\quad \bigcirc(display(compressor_error_icon)\mathcal{U} \\ &\quad (select(compressor_error_icon) \vee state(compressor, working)) \end{aligned} \tag{5}$$

This clause states that a failure in a compressor is acknowledged if in the present interval the compressor has failed and in the next interval a warning icon is displayed for that component until the user selects that icon or the compressor is working again.

It is important to emphasise that the lack of status information does not just cause problems for the refinement of the $\Diamond$ operator. Similar comments can be made about the input and output abstractions in (3), (4) and (5). These requirements are not at a level of detail where a development team can proceed towards implementation. For instance, mice or tracker balls might be used to satisfy these requirements. This decision will have a profound impact upon anticipated error rates and hence upon the protection mechanisms that must be used to guard against inadvertent selection [11]. As in our earlier example, there is no means of identifying the need for further refinement from the formal specification alone.

Many theoretical works on formal methods ignore the pragmatic problems that arise from the refinement of large scale specifications [26]. There is an assumption that the level of detail will gradually increase in a uniform manner. Recent research into design practice has revealed that such assumptions are unwarranted [21, 16]. Less critical areas of an interface may be left at a high level of abstraction. More important areas and, in particular, those aspects of an interactive system that preserve the safety of an application must be considered in greater detail. Unless these areas can be agreed upon then development activities may be needlessly duplicated or omitted. This problem is exacerbated by the fact that different groups may be working on different areas within a specification. The lack of status of information for abstract clauses, such as (3), (4) and (5), can prevent designers from coordinating and focusing their efforts upon those areas of a specification that require the most work.

4.4 Lack Of Alternatives

Perhaps the biggest problem in applying formal methods is that they do not capture the range of alternative options that are considered but rejected during the development of safety-critical systems. This is a significant problem because regulatory and commercial organisations require proof that designers have selected the correct approach from the available alternatives [7]. The absence of any explicit representation for rejected alternatives also creates problems for the team based development of safety-critical user interfaces. For instance, one group might consider the use of a command line dialogue style without understanding why another group chose to use the graphical techniques in (3). This could potentially wreak havoc in the design of large-scale systems. This confusion arises because there is no means of representing alternative options within a specification without according them the same status as those options that are actually to be exploited within a final implementation:

$$user_command_line_acknowledge(compressor_failure) \Leftarrow$$
$$state(compressor, failed) \wedge$$
$$\bigcirc(display(compressor_error_icon) \wedge$$
$$\Diamond(input(\text{“}confirm\ compressor\ error''})) \quad (6)$$

This clause states that a failure in a compressor is acknowledged if in the present interval the compressor has failed and in the next interval a warning icon is displayed for that component and eventually the user types “$confirm\ compressor\ error''$.

The representation of design alternatives is of critical importance during the development of complex interfaces to real-world applications. Revisions to a set of initial requirements frequently force designers to re-consider the options that were discarded early in the development cycle. For instance, the decision to sub-contract the development of control panels might force designers to exploit keyboards rather than pointing and selection devices. Under such circumstances, designers might be forced to re-consider the option described by (6) rather than the graphical interaction style assumed in (3).

5 Literate Specifications

Design rationale techniques have, typically, exploited semi-formal, graphical notations. Shum argues that this provides the flexibility that is required to represent the many different forms of decisions that have to made during the life-cycle of an interactive system [32]. A further advantage is that designers are not forced to learn complex syntax and composition rules [16]. Klein argues that this the flexibility and informality of design rationale supports the communication of design decisions between the members of concurrent development teams [21]. These benefits can be recruited to avoid the limitations that restrict the use of formal methods in the development of large scale, safety-critical user interfaces. Figure 2 provides an overview of our approach to the literate specification of human computer interfaces. Formal

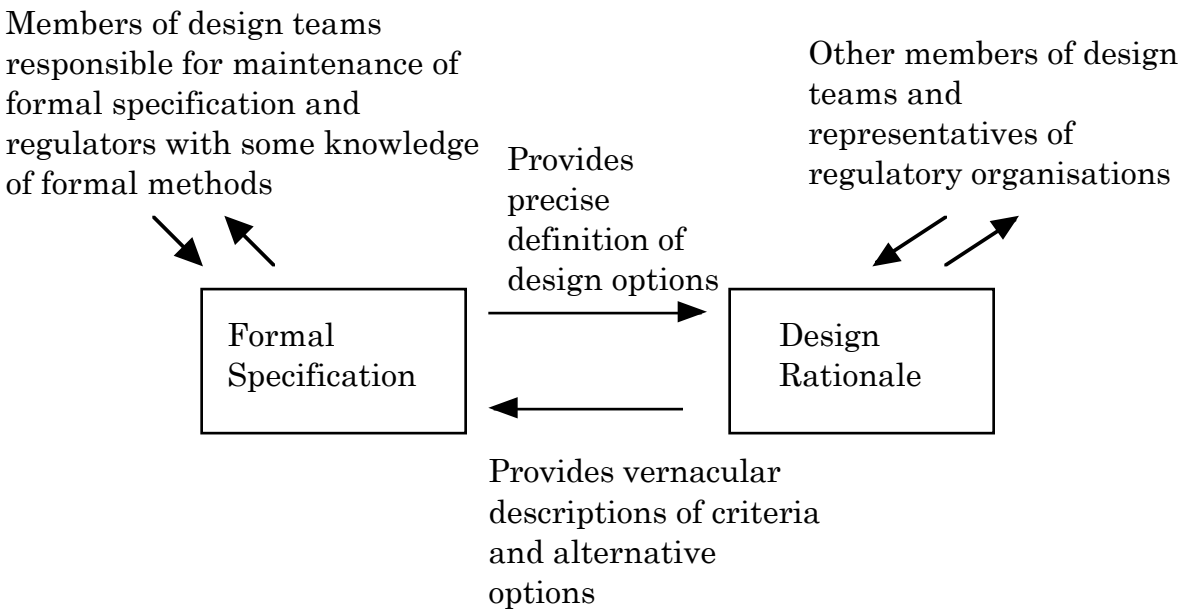

Figure 2: Overview Of The ‘Literate’ Specification Technique

specification techniques are intended to provide the level of precision and rigour that is demanded during the development of safety-critical systems. Design rationale is intended to provide the degree of flexibility that is required in order to communicate the justifications that lie behind particular design options.

5.1 Lack Of Vernacular Descriptions

The first way that design rationale might be recruited to support literate specifications is by **explicitly** linking vernacular and semi-formal argumentation structures to the abstract clauses of formal methods. For instance, Figure 3 illustrates how a QOC rationale might be used to support the automatic cancellation of blow-back warnings once an error has been resolved. The requirement that the warning should be

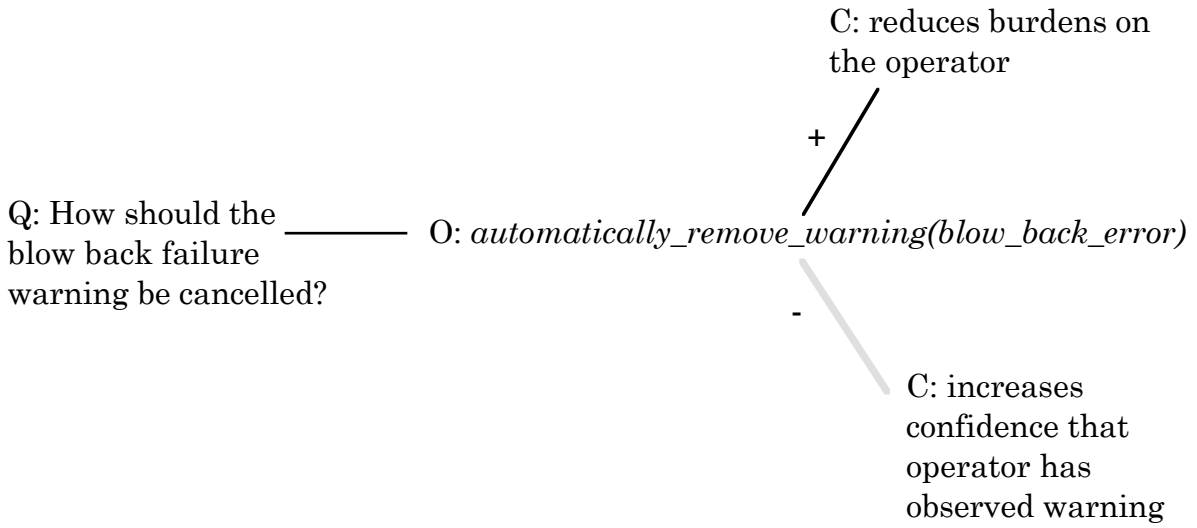

Figure 3: Linking Vernacular Argumentation And Formal Requirements

cancelled automatically is supported by the argument that this will reduce the burdens on the operator. It is not supported by the argument that operators will be more likely to observe the warning. It is intended that the formal clauses and semi-formal rationale will be regarded as the co-products of our 'literate' approach to specification. Although some training in discrete mathematics may still be required to understand the precise requirements in a formal specification, it is now possible for all members of a design team to question the arguments that support these formal assertions.

5.2 Lack Of Justification

An important motivation behind 'literate' specifications is the need to **explicitly** represent the justification for particular requirements. Figure 4 provides a further illustration of this technique. It is intended that these diagrams should be produced as a co-product of the formal development process. As various alternatives are assessed, the justification for each design option should be explicitly represented in a corresponding QOC diagram. It would then be possible to trace the reasons for adopting a particular requirement within a human-machine interface. It is important to acknowledge that there will be an additional overhead in developing both a design rationale and a formal specification. Following MacLean, Young, Belotti and Moran we argue that our integrated approach will repay any initial investment by "(a) providing an explicit representation to aid reasoning about the design and the consequences of changes to it, and (b) serving as a vehicle for communication, for example, among the members of a design team or between the original designers and the later maintainers of a system." [23].

5.3 Lack Of Status Information

It can be difficult to determine which parts of a formal specification require further analysis. The integration of mathematical requirements and semi-formal, design rationale can be used to explicitly introduce this status information into abstract specifications of user interfaces. For instance, clause (3) used the $\Diamond$ (read as 'eventually') operator to represent the delay that can arise before a user selects the compressor warning. This formula does not specify the duration of the interval. This is unsatisfactory because at some point additional presentation resources may be required in order to direct the operator's attention towards the warning. If those areas of a specification that require further refinement cannot be easily identified then critical design problems may be overlooked during subsequent development. In larger scale

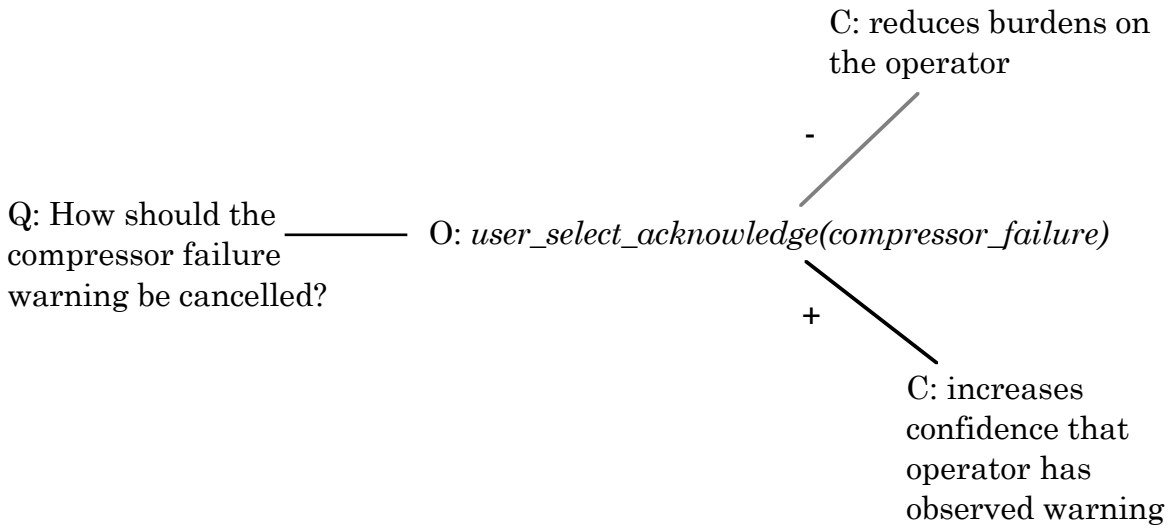

Figure 4: Justifying Formal Requirements Using Design Rationale

projects, these partially refined clauses may be lost amongst a mass of other detail. It is important to emphasise that this problem is not simply related to temporal operators such as $\Diamond$. It also affects the wider use of abstractions in formal specifications. For instance, similar comments could be made about the need to refine graphical terms such as the *compressor_error_icon* into individual lines, fonts etc. These details must be introduced before abstract specifications can be used to implement human-machine interfaces [16]. Many of these problems can be avoided by using design rationale to explicitly represent the status of abstract clauses. By this we mean that QOC diagrams can be used to identify those options that require further work. Figure 5 illustrates this technique. In this diagram, the option that (3) should be used to cancel the compressor failure is opposed by the observation that the clause does not consider the additional alarms which must be provided if the user does not observe the initial warning. The use of the box around the criterion is not a standard part of the QOC syntax and is simply included for emphasis. A number of benefits can be gained from the introduction of status information into formal specifications. In particular, it supports the use of mathematical requirements within concurrent engineering teams. For instance, clauses such as (3) have implications both for systems engineers and for interface designers. The former group must allocate sufficient sensing devices to detect the failure. The latter group must devise appropriate presentation techniques to display the warning. By explicitly representing the status of this option both parties can recognise that this clause requires further work. In this way, design rationale can provide a focus for the subsequent development of literate specifications.

5.4 Lack Of Alternatives

The integration proposed in previous sections can be extended to represent alternative options within a mathematical specifications. Traditionally, formal methods have been used to provide a single representation of system behaviour. This is unsatisfactory. Design progresses by selection appropriate options from a range of alternatives. It is frequently the case that changes in a design or in a set of customer requirements will force designers to reconsider those options that had been rejected during earlier stages of development. As before, the integration of formal specification techniques and design rationale notations can be used to address this problem. For instance, Figure 6 uses the extended QOC in our literate specification to represent alternative options for the confirmation of the compressor failure icon. This diagram shows how a number of clauses might be represented as different design options within a QOC diagram. From this figure, designers can construct the arguments for and against command lines and icon selection as means of confirming compressor warnings. The former technique suffers from keystroke errors while the latter does not. This is a simplistic analysis but it illustrates the general point that design rationale can be used to represent the alternative options that may remain implicit within conventional approaches to formal specification. Figure 6 also illustrates the need for tools to support our 'literate'

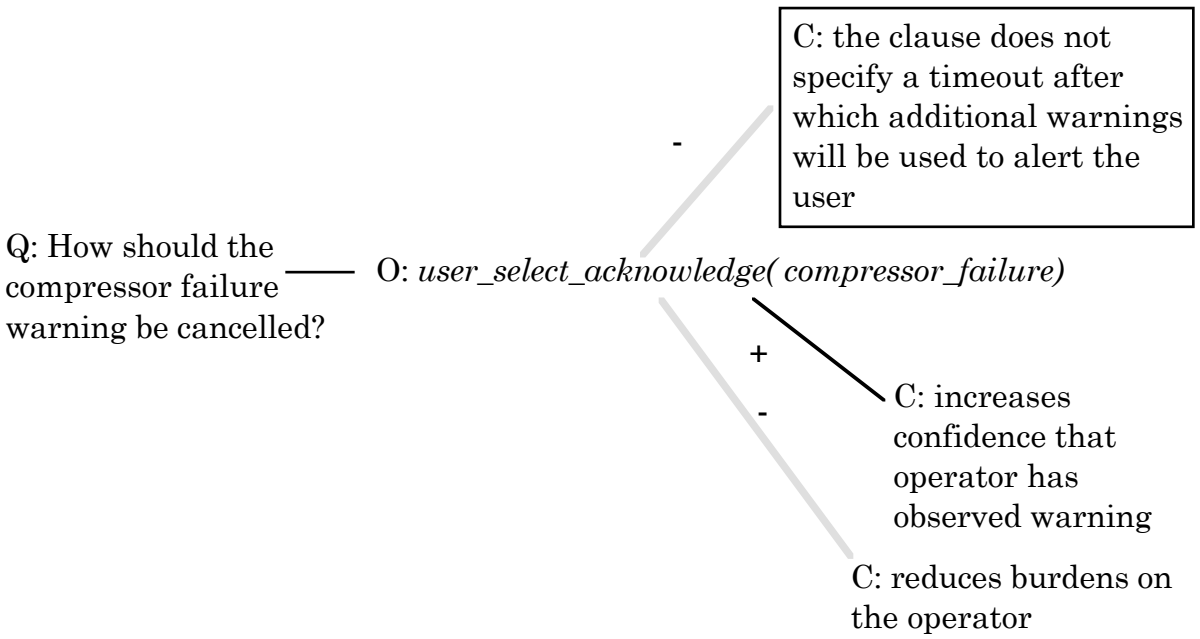

Figure 5: Representing Status Information Using Design Rationale

specifications. The links between various options and criteria form a complex web of inter-connections.

6 Tool Support

We have developed two sets of tools that can be used to support the integration of formal methods and design rationale in our literate approach to interface specification. Together with Chan we have implemented a direct manipulation editor for QOC diagrams [3]. The option nodes in this tool can be annotated with the elements of a formal specification. A limitation with this approach is that the body of the specification has to be extracted using the browsing facilities of the environment. Current work is extending this tool so that designers can access the nodes of the QOC directly from the textual representations of the logic specification. This represents an extension of the literate programming techniques that were first proposed by Knuth [22] and later developed by Thimbleby [35]. These approaches used hypertext as a means of linking natural language documentation to the listings of computer programs. Similarly, our tool links formal requirements to semi-formal and vernacular documentation.

The second tool has been built around the Prelog logic programming environment [18]. This enables designers to reason about the content and the justification for formal specifications. The following clauses illustrate the use of this tool by representing the different design options for the cancellation of the compressor warning:

```
qoc(compressor_cancel, user_select_acknowledge, confirms_observation).
qoc(compressor_cancel, user_select_acknowledge, not(less_screen_clutter)).
qoc(compressor_cancel, automatically_remove_warning, less_screen_clutter).
qoc(compressor_cancel, automatically_remove_warning, not(confirms_observation)).
```

Previous tools for design rationale provide little more than hypertext browsers and structure editors. In contrast, designers can use Prelog's resolution proof techniques to query the structure of a QOC document. For example, this approach can be used to establish what criteria are used to support an option within any of the many hundreds of QOC diagrams that might be required for a large scale user interface. This would require significant resources of time and patience if designers were forced to do this manually using the browsing facilities of most design rationale tools:

```
|?- qoc(compressor_cancel,user_select_acknowledge, Criteria).
```

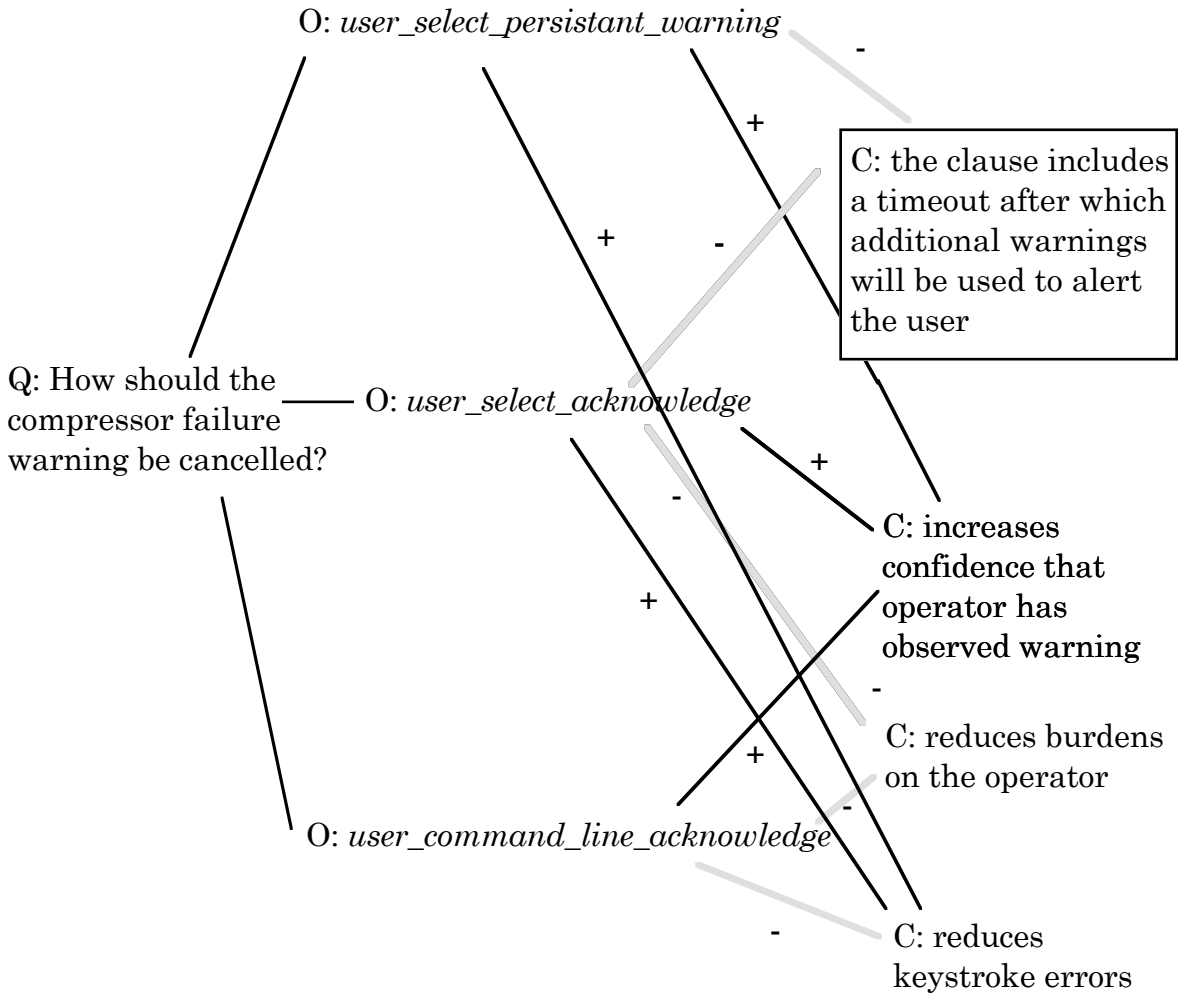

Figure 6: Representing Alternative Options Using Design Rationale

```
Criterion = confirms_observation ?
Criterion = not(less_screen_clutter) ?
```

The clauses presented above can also be used to automatically generate the graphical form of the QOC notation using Prelog's presentation facilities. Figure 7 illustrates the architecture of this system. The logic clauses are translated by the Tokio programming environment into the low level presentation commands that are required to reproduce the QOC diagrams on particular interaction devices. Prelog was originally developed as prototyping tool for temporal logic specifications. It, therefore, provides a common environment that can be used to support both the formal and informal components of our literate approach to interface specification.

7 Conclusion And Further Work

In this paper we have argued that formal specification techniques cannot easily be used to support the large-scale development of safety critical interfaces. It is argued that the lack of vernacular descriptions prevents many designers from accessing the products of formal analyses. Mathematical specification techniques have also been criticised because they do not represent the arguments that lie behind critical design decisions. This makes it difficult for designers to justify their development decisions to external, regulatory authorities. A further problem is that formal methods, typically, do not represent the status of

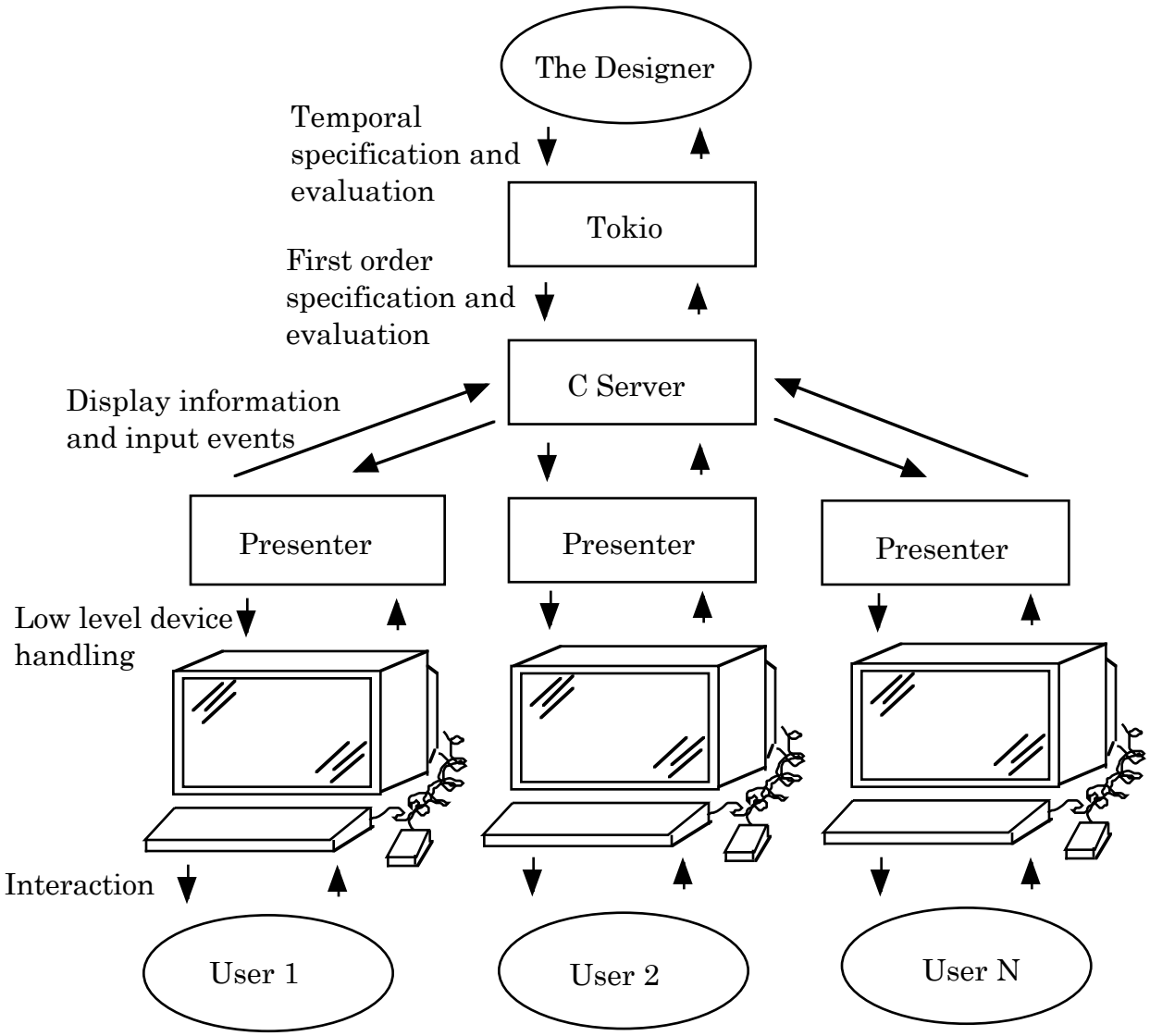

Figure 7: The Prelog Architecture

abstract requirements. In large scale systems, this makes it difficult for teams of designers to coordinate their efforts around those areas of an interface that require the most work. Finally, formal specifications are constructed to provide an unambiguous representation of a design. They do not represent the alternative options that were considered but rejected as development progresses. This creates practical problems if these options must be reconsidered during subsequent stages of design. This is a significant limitation because user and customer requirements frequently change during the development of interactive systems. This paper has shown that all of these limitations can be addressed by representing clauses as the options in the QOC notation. This provides the status information, the justification and the alternative options that often remain implicit during the formal development process.

There are a number of areas for further work. Fischer, Lemke, McCall and Morch [10] argue that it is possible to develop a large corpus of domain specific design rationale to guide the development of future systems. This approach might be extended to support our use of QOC within literate specifications. Libraries of formal requirements and informal rationales might reduce the costs of future projects through the re-use of design documents. Such applications are a long term objective. Further work is required to determine whether it is possible to represent more generic clauses in the formal components of our

rationales. The examples that have been used to illustrate our techniques are highly specific to particular classes of safety-critical interfaces.

In this paper we have focussed upon the benefits that design rationale can bring to formal methods. There are also a number of benefits that can be obtained through the use of more formal techniques within design rationale. For instance, the `qoc` clauses that are supported by the Prelog environment can automatically be checked to ensure consistency. In other words, designers can perform queries to ensure that criteria are not used both to strengthen and weaken an option in different contexts. Such checks are necessary if design rationales are to stand up to inspection by regulatory authorities. Brevity has prevented a more detailed discussion of these techniques. It is important to emphasise, however, that the advantages are only gained at a cost in terms of the expressiveness of the rationale. The Prelog clauses, represented in the previous section, lose many of the vernacular qualities that provide design rationale with its great flexibility. Future work intends to investigate whether some compromise is possible between these two approaches. For instance, consistency checks may still be performed using structural induction over semi-formal QOC diagrams.

References

[1] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *ACM Symposium On The Theory Of Computing*, pages 51–63, New York, United States of America, 1985. ACM Press.

[2] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, pages 189–209, July 1993.

[3] T. Chan. A graphical tool to support the QOC notation. Technical report, Department of Computer Science, University of York, York, United Kingdom, 1994. M.Sc thesis.

[4] K.L. Clark. The synthesis and verification of logic programs. Technical Report 81/36, Department of Computing, Imperial College Of Science And Technology, London, United Kingdom, 1981.

[5] J. Conklin and M.L. Begeman. gIBIS: A tool for all reasons. *Journal Of The American Society For Information Science*, pages 200–213, May 1989.

[6] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, U.S. Department of Commerce, National Institute Of Standards And Technology, Githersburg, United States of America, 1993.

[7] K.R. Davies. Techniques for the identification and assessment of major accident hazards. In J. Burgoyne, editor, *The Assessment And Control Of Major Hazards*, pages 289–308. Pergamon Press, Oxford, United Kingdom, 1985.

[8] A.J. Dix. *Formal Methods For Interactive Systems*. Academic Press, London, United Kingdom, 1991.

[9] R.E. Fields, P.C. Wright, and M.D. Harrison. A task centred approach to analysing human error tolerance requirements. In P. Zave and M.D. Harrison, editors, *Second IEEE International Symposium On Requirements Engineering*, pages 18–26. IEEE Press, Los Alamitos, United States of America, 1995.

[10] G. Fischer, A.C. Lemke, R. McCall, and A.HJ. Morch. Making argumentation serve design. *Human Computer Interaction*, 6(3 and 4):393–419, 1991.

[11] I.A.R. Galer and B.L. Yap. Ergonomics in intensive care: Applying human factors to the design and evaluation of a patient monitoring system. *Ergonomics*, 23(8):763 – 779, 1980.

[12] C.W. Johnson. Applying temporal logic to support the specification and prototyping of concurrent multi-user interfaces. In D. Diaper and N. Hammond, editors, *People And Computers VI: Usability Now*, pages 145–156. Cambridge University Press, Cambridge, United Kingdom, 1991.

[13] C.W. Johnson. A probabilistic logic for the development of safety-critical interactive systems. *International Journal Of Man-Machine Studies*, 39(2):333–351, 1993.

[14] C.W. Johnson. Specifying and prototyping dynamic human-computer interfaces for stochastic applications. In J.L. Alty, D. Diaper, and S. Guest, editors, *People And Computers VIII*, pages 233–248. Cambridge University Press, Cambridge, United Kingdom, 1993.

[15] C.W. Johnson. FDR - a formal design rationale. Technical report, Department of Computing Science, University of Glasgow, Glasgow, United Kingdom, 1995.

[16] C.W. Johnson. Using Z to support the design of interactive, safety-critical systems. *To appear in IEE Software Engineering Journal*, 1995.

[17] C.W. Johnson and P. Gray. A critical analysis of interface specification notations. Technical Report TR-1995-6, Department of Computer Science, University of Glasgow, Glasgow, Scotland, G12 8QQ., 1995. Early Version Of Paper To Appear In The Second Eurographics Workshop On The Design, Specification And Verification of Interactive Systems.

[18] C.W. Johnson and M.D. Harrison. Declarative graphics and dynamic interaction. In F.H. Post and W. Barth, editors, *EUROGRAPHICS '91*, pages 195–207. Elsevier Science Publications, North Holland, Netherlands, 1991.

[19] C.W. Johnson and M.D. Harrison. Using temporal logic to support the specification and prototyping of interactive control systems. *International Journal Of Man-Machine Studies*, 36:357–385, 1992.

[20] C.W. Johnson, J.C. McCarthy, and P.C. Wright. Using a formal language to support natural language in accident reports. *Ergonomics*, 38(6):1265–1283, 1995.

[21] M. Klein. Capturing design rationale in concurrent engineering teams. *IEEE Computer*, 26(1):39–47, 1993.

[22] D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[23] A. MacLean, R.M. Young, V.M.E. Bellotti, and T.P. Moran. Questions, options, and criteria: elements of design space analysis. *Human-Computer Interaction*, 6(3-4):201–50, 1991.

[24] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J. Strother Moore, editors, *The Correctness Problem In Computer Science*, pages 215–273. Academic Press, London, United Kingdom, 1981.

[25] Ministry Of Defence. *Requirements for the Procurement of Safety Critical Software*, MOD DEF-STAN 00-55, London, United Kingdom, 1991.

[26] C. Morgan. *Programming From Specifications*. Prentice Hall, London, United Kingdom, 1990.

[27] B. Moszkowski. Some very compositional temporal properties. Technical Report TR-466, Department of Computing Science, University of Newcastle Upon Tyne, Newcastle, United Kingdom, 1993.

[28] National Aeronautic and Space Administration. *Advanced Orbiting Systems - Architectural Specification For The CCSDS Secretariat*, Washington DC, United States of America, 1989.

[29] J. Rasmussen and M. Lind. Coping with complexity. Technical Report Risø-M-2293, The Risø National Laboratory, Roskilde, Denmark, 1981.

[30] J. Rasmussen and W.B. Rouse. *Human Detection And Diagnosis Of System Failures*. Plenum Press, New York, United States Of America, 1981.

[31] J. Reason. *Human Error*. Cambridge University Press, Cambridge, United Kingdom, 1990.

[32] S. Buckingham Shum. Cognitive dimensions of design rationale. In D. Diaper and N.Hammond, editors, *People And Computers VI: Proceedings Of HCI'91*. Cambridge University Press, Cambridge, United Kingdom, 1991.

[33] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, London, United Kingdom, 2 edition, 1989.

[34] B. Sufrin and J. He. Specification, refinement and analysis of interactive processes. In M. D. Harrison and H. W. Thimbleby, editors, *Formal methods in Human Computer Interaction*, pages 153–200. Cambridge University Press, Cambridge, United Kingdom, 1990.

[35] H.W. Thimbleby. *User Interface Design*. Addison Wesley, Wokingham, United Kingdom, 1990.

[36] R. Took. Integrating inheritance and composition in an objective presentation model for multiple media. In F.H. Post and W. Barth, editors, *EUROGRAPHICS '91*, pages 291–303. Elsevier Science Publications, North Holland, Netherlands, 1991.

[37] N. Worley and J. Lewins, editors. *The Chernobyl Accident And Its Implications For The United Kingdom - Report Number 19 Of The Watt Committee on Energy*. Elsevier Applied Science, London, United Kingdom, 1988.

[38] K.C.B. Yakemovic and J. Conklin. Report on a development project use of an issue based information system. In *Proceedings of CSCW'90: Computer Supported Cooperative Work*, pages 105–118, New York, United States of America, 1990. ACM Press.