9

# Expanding the Role of Formal Methods in CSCW

**CHRIS JOHNSON**
*University of Glasgow*

**ABSTRACT**

Before we can build CSCW systems it is important to have a clear idea of the requirements that they must satisfy. This chapter argues that formal methods can be used to help represent and reason about these requirements. Unfortunately, the formal notations that support the development of single-user interfaces cannot easily be used to support the design of multi-user applications. Traditional approaches abstract away from the temporal properties that characterize interaction with distributed systems. They often neglect the input and output details that have a profound impact upon multi-user interfaces. The following pages argue that these details can be integrated into formal specifications. For the first time, it is shown how mathematical specification techniques can be enhanced to capture physical properties of working environments. This provides a link between the physiological studies of ergonomics and the interface design techniques of HCI. Such links have been completely neglected within previous work on design notations. In all of this, the intention is to fight against a narrow, myopic, view of formal methods. These notations need not simply be used to focus in upon a relatively small number of software engineering principles. The aim is to show that formal methods can be used creatively to solve a vast range of design problems within complex multi-user interfaces.

## 9.1 INTRODUCTION

The term “formal method” is used to refer to a variety of notations and development techniques that support the rigorous development of complex systems. By the term rigorous, we mean that they have a mathematical basis which can be used to determine whether a particular description of a complex system is in some sense correct. At first sight, the use of

*Computer Supported Cooperative Work,* Edited by Beaudouin-Lafon
© 1999 John Wiley & Sons Ltd

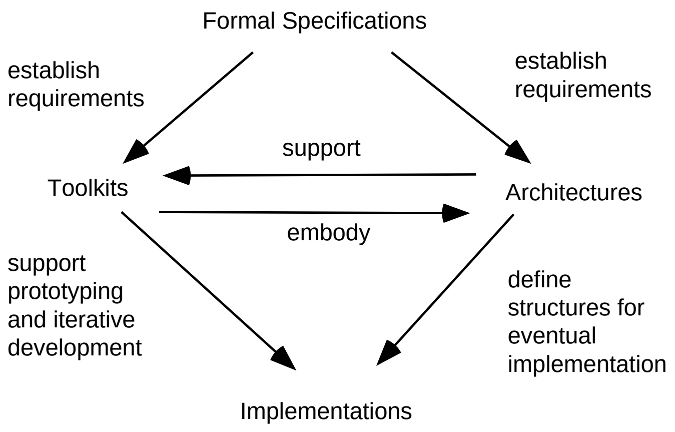


**Figure 9.1** Formal methods and the development of CSCW systems

abstract mathematics may seem to have little connection with the previous chapters in this collection. These have focused upon particular CSCW interfaces, development architectures or multi-user toolkits. In contrast, this chapter argues that formal methods offer considerable benefits for the development of CSCW systems. Figure 9.1 illustrates how they might guide the different approaches described in the previous chapters of this book. Before designers can select appropriate architectures, they must have a clear idea of the requirements that their system must satisfy. Before development teams can identify potential toolkits, they must first establish the constraints that their interface must satisfy.

### 9.1.1 Why Use Formal Methods In CSCW Systems?

There are a number of additional, commercial reasons why formal methods are being recruited to support the design of CSCW systems. Mathematical notations are increasingly being used in the development of large-scale applications. Craigen, Gerhart and Ralston's survey for the US Department of Commerce cites projects ranging from nuclear reactor control systems to French rapid transport applications [Cra93]. Formal methods have also been used to support the development of interactive systems [Joh96b]. A number of authors have extended these techniques to support the design of multi-user interfaces. For instance, Palanque and Bastide have developed a graphical notation to represent simultaneous transactions by multiple users on shared interaction objects [Pal95]. The applications, cited above, all focus upon CSCW interfaces for office-based applications. Safety-critical systems, perhaps, represent the greatest potential for the application of formal methods. Johnson, McCarthy and Wright have exploited a range of graphical formalisms to identify human factors problems amongst the aircrews in several major accidents [Joh94a]. A common motivation behind all of this work has been a concern to avoid some of the weaknesses that natural language presents for the development of multi-user systems [Joh95a].

### 9.1.2 The Limitations of Natural Language

What is a formal method? In one sense, a formal method is any notation that has a clear syntax and a well-understood semantics. By syntax, we mean that there are rules for building up sentences out of simpler components. For instance, the sentence “all users can quit the system at any time” follows the established grammatical rules for the English language. “ Time users all any system the quit can at the” breaks the rules. The term semantics is used to refer to the meaning of a sentence. We can all hopefully agree upon the intended meaning of the first example. If we break the syntactic rules, as in the second example, then it is more difficult to extract the meaning of a sentence.

According to our definition, natural language is a formal method. The previous paragraph has shown that it has both a syntax and semantics. Without these underlying properties it could not be used to support the development of multi-user systems. Designers and engineers would not be able to interpret phrases such as “all users can quit the system at any time”. Unfortunately, there are a number of problems. The intended meaning of an English sentence is not always clear. For instance, the previous example does not describe the input devices and command sequences that each user might exploit to quit the system. Such ambiguity may cause irritation and inconvenience in the design of collaborative working environments. In safety-critical applications, the consequences can be much more profound. For example, the following recommendation was published by the United Kingdom’s Department of Energy in the aftermath of the Piper Alpha accident:

> “ There should be a system of emergency exercises which provides Offshore Installations Managers with practice in decision-making in emergency situations, including decisions on evacuation. All of the Offshore Installations Managers and their deputies should participate regularly in such exercises” [Cul90, page 399, para 20.61].

These natural language requirements cannot easily be used to support the detailed development of CSCW systems. They do not provide enough information about the “emergency exercises” for designers to review existing practice. It is ambiguous in the sense that any two individuals might disagree about what is meant by an “emergency situation”. These problems provide real barriers to the use of natural language in the team-based development of CSCW systems.

### 9.1.3 The State of the Art

Formal notations help to reduce the ambiguity and imprecision that characterize natural language. This is, typically, done by imposing constraints upon the sentences that are valid within a language. For example, syntactic rules can be used to define a structure or format for natural language clauses. Within this general approach, there are a range of different ways in which formal notations can support the development of CSCW systems.

#### 9.1.3.1 Formal Methods for Principled Design

An important benefit of formal methods is that they enable designers to express important properties of CSCW systems at an extremely high level of abstraction. By stripping out low-level implementation details, it is possible to focus in upon common properties that affect a large number of multi-user systems. For example, Dix, Rodden and Somerville use the following formulae to specify the notion of fidelity in a multi-user version control system [Dix97]. By fidelity, they intend that the version history for any object should accurately reflect the transactions that have been performed upon it. In the following, @(context) is an actual context corresponding to a context label in a version manager:

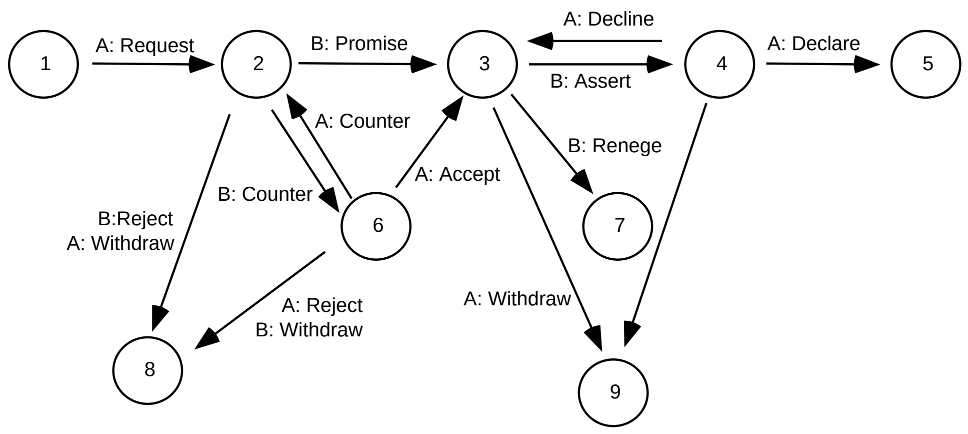


**Figure 9.2** Conversation for action (from Winograd and Flores)

$$\begin{aligned} &\forall context \in Contexts, context' \in\ dom\ world\ context(entity')\ : \\ &\quad world\ context(entity')(context') = world@(context')(entity) \end{aligned} \tag{9.1}$$

The important point here is that mathematical abstractions, such as the set of $Contexts$, can be used to represent the concept of fidelity without referring to the particular details that must be considered during a full implementation. The use of mathematics encourages designers to carefully formulate an explicit expression or representation for such properties. This avoids the misunderstandings that can arise when such high-level goals are left as implicit objectives for design teams.

### 9.1.3.2 Formal Methods for Interaction Architectures

The previous section briefly argued that formal notations can be used to represent high-level design objectives. They can, however, also be used to direct the implementation of particular systems. Winograd and Flores exploited this approach when using state transition diagrams to develop a high-level architecture for their Coordinator application [Win87]. Figure 9.2 shows how the states, denoted by circles, are used to represent critical points in a “conversation for action”. The transitions between states, denoted by arcs, are used to represent communication between the participants. The key point here is that the syntactic structures of the notation help designers to strip aside the mass of irrelevant detail that can obscure critical properties of CSCW systems. By focusing on states and the transitions between them, the previous diagram clearly illustrates the various opportunities that face each participant at each stage of an interaction. Figure 9.2 also illustrates some of the weaknesses of this approach. State transition diagrams provide a very sequential view of interaction with CSCW systems. It can also be difficult to capture some of the detailed cognitive and system factors that affect interaction with multi-user applications.

| User 1 | | | User 2 | | | Computer/communications infrastructure | |
|---|---|---|---|---|---|---|---|
| location | internal actions | user (articulatory) actions | location | internal actions | user (articulatory) actions | perceivable computer actions | internal actions |
| Sector A | locate button | select button | | | | button hilited on 1 | user 1's request sent |
| | | | | | | user 2's machine shows request pending | |
| | | | Sector B | observe request pending symbol | select view request menu item | | |
| Sector C | | | | | | "re-connecting" message on 1 | dispatch transfer request from 2 |

**Figure 9.3** UAN showing interaction over a mobile network

### 9.1.3.3 Formal Methods for Task Analysis in Traces of Interaction

The semi-formal User Action Notation (UAN) avoids some of the limitations of state transition diagrams [Hix93]. UAN organizes the actions comprising a task into categories based on the agent that executes them and their function in the task. These categories define the syntax of the notation and are represented as the columns of a tabular format. For example, Figure 9.3 shows how an extended form of UAN can be used to analyze mobile communication between concurrent users of a multiple computer system [Joh97]. Initially, user 1 is in cell A and requests information from user 2. The communications infrastructure forwards the request to user 2 who views it before dispatch. In the meantime, user 1 has moved into another cell and the system must reestablish their connection through another transceiver. It is important to emphasize that Figure 9.3 represents a different application of formal methods from that shown in Figure 9.2. Rather than representing a high-level architecture for interaction, as in the case of Coordinator, the UAN diagram is being used to represent and reason about particular user tasks during a particular trace of interaction. Unfortunately, a number of problems limit the utility of this notation. It provides no means of reasoning about temporal properties. This is important because the handover delay in Figure 9.3 might have a minimal effect if it lasted a few seconds. If it took several minutes then the "re-connecting" message might have to be reworded to provide more information about the cause of the delay. Temporal information can be represented using the extended XUAN notation [Gra95]. The more general point here is again that the restricted syntax of formal notations helps designers to focus in on critical properties of a CSCW system. In Figure 9.3, those properties include the physical locations of the users and their observable actions. However, this is only achieved at the cost of other properties, such as temporal relationships, that cannot be so easily captured within the syntactic structures.

### 9.1.3.4 Formal Methods for Accident Analysis

In contrast to the tabular form of UAN, Petri Nets provide a graphical notation that has long been used to represent temporal properties of interactive systems [Kra91]. Bastide and Palanque [Bas90] exploit Petri Nets to derive formal specifications of interactive systems at a very high level of abstraction. Johnson et al [Joh94a] have shown that Petri Nets can be used to represent the operator–system interaction which can lead to accidents in safety-critical sys-

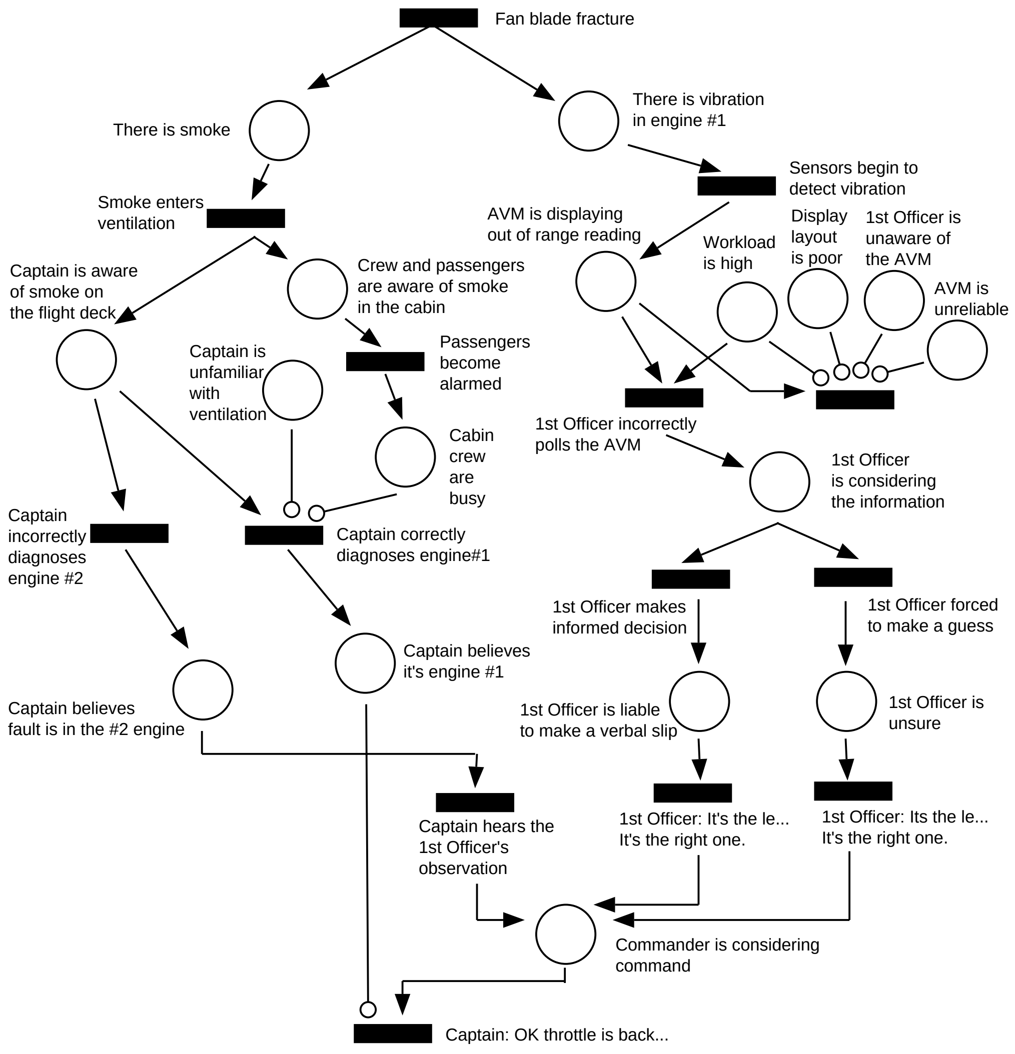

Fan blade fracture
There is smoke
There is vibration in engine #1
Sensors begin to detect vibration
Smoke enters ventilation
AVM is displaying out of range reading
Display layout is poor
1st Officer is unaware of the AVM
Workload is high
AVM is unreliable
Captain is aware of smoke on the flight deck
Crew and passengers are aware of smoke in the cabin
Passengers become alarmed
Captain is unfamiliar with ventilation
Cabin crew are busy
1st Officer incorrectly polls the AVM
1st Officer is considering the information
Captain incorrectly diagnoses engine #2
Captain correctly diagnoses engine#1
1st Officer makes informed decision
1st Officer forced to make a guess
Captain believes it's engine #1
Captain believes fault is in the #2 engine
1st Officer is liable to make a verbal slip
1st Officer is unsure
Captain hears the 1st Officer's observation
1st Officer: It's the le... It's the right one.
1st Officer: Its the le... It's the right one.
Commander is considering command
Captain: OK throttle is back...

**Figure 9.4** A high-level Petri net

tems. Timing properties are represented by sequences of places. These are denoted by a circle and can be used to show states of interaction. Places are linked by transitions. These are denoted by rectangles and can be used to represent events during interaction. Figure 9.4 illustrates this approach. It also shows how Petri Nets can be used to analyze relatively complex traces of group interaction. Once again, it is important to emphasize that the previous diagram represents a different style of application for formal methods. Previous examples have used state transition diagrams to analyze high-level architectures for CSCW systems, UAN was used to analyze user tasks during a potential trace of interaction. Here, Petri Nets are being used to analyse crew interaction prior to the Kegworth air crash. This more situated use of a notation helps to focus in upon critical features of a previous failure as a means of establishing requirements for future systems. There are, however, a number of limitations that restrict the utility of Petri Nets for the design of CSCW systems. For example multi-user undo cannot easily be represented using this notation [Gra95]. Similarly, the associated proof techniques that support this approach can be surprisingly complex given the intuitive appeal of the graphical representation.

### 9.1.3.5 Formal Methods for Proof

The ability to prove properties of a system, prior to implementation, is a key benefit of formal methods. Proof is essentially a form of reasoning or argument that uses the syntactic rules of a notation to determine the validity of a theorem or hypothesis. This can be illustrated by the following example using first-order logic. Designers might specify that a system should be shut down if two users issue input to that effect:

$$\begin{aligned} &shut\_down \Leftarrow \\ &\quad input(user\_1, stop) \wedge input(user\_2, stop). \end{aligned} \tag{9.2}$$

The system is shut down if $user\_1$ and user_2 issues input to stop the system.

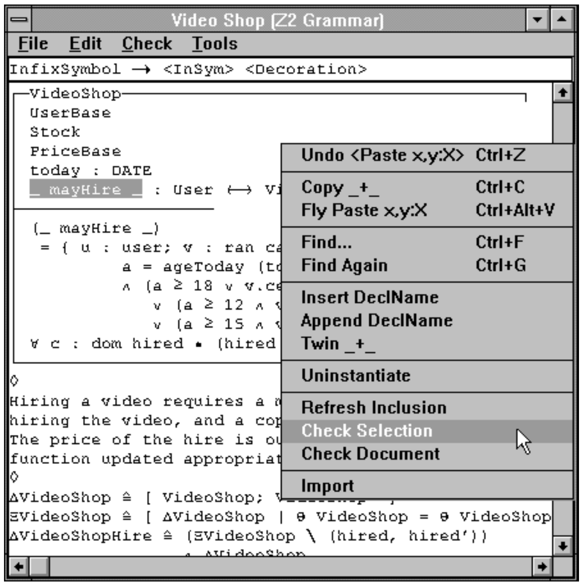
First-order logic provides a proof rule which states that if we know that some fact $P$ is implied by some other fact $Q$ and we know $Q$ then it is safe to conclude $P$:

$$P \Leftarrow Q, Q \vdash P. \tag{9.3}$$

Given that $P$ is true if $Q$ is true and we know $Q$ then it is safe to conclude that $P$ is true.

Given the two previous clauses we can now establish whether or not the system will ever be shut down. In a model checking approach to theorem proving, this would be achieved by generating possible states of the system and inspecting those states to determine whether or not both users had issued the appropriate input. This illustrates an important weaknesses of theorem proving for interactive systems. There is no automatic means of determining whether or not users will actually provide the anticipated $input$ in any state of the system. On the other hand, this approach does force designers to consider the assumptions that they make about operator behavior. For instance, the proof process outlined in (9.3) forces designers to consider those situations in which users might be expected to cooperate in the manner described by (9.2).

A range of tools support the application of formal methods. For example, theorem proving tools provide designers with a semi-automatic means of checking whether certain properties do or do not hold for a particular design [Har95]. Similarly, model checking tools can be used to search for particular situations that may or may not arise during the course of interaction. These tools increase the level of automation provided by theorem proving systems and provide direct means of searching for particular scenarios of interaction. Other tools can also be recruited to aid the formal development of multi-user systems. For example, Figure 9.5 illustrates the user interface to Logica's commercial Z tool, called Formaliser. This automatically helps designers to construct syntactically correct specifications through structure editing. Other tools can be used to "directly" develop prototype implementations from formal specifications [Joh92]. This is important because mathematical specification techniques provide an extremely poor impression of what it would be like to interact with a potential interface.



```
InfixSymbol → <InSym> <Decoration>
┌VideoShop──────────────────────────────
  UserBase
  Stock
  PriceBase
  today : DATE
  _ mayHire _ : User ⟷ Vi
├──────────────
  (_ mayHire _)
   = { u : user; v : ran ca
            a = ageToday (to
            ∧ (a ≥ 18 ∨ v.ce
               ∨ (a ≥ 12 ∧ v
               ∨ (a ≥ 15 ∧ v
  ∀ c : dom hired • (hired
◊
Hiring a video requires a n
hiring the video, and a cop
The price of the hire is ou
function updated appropriat
◊
ΔVideoShop ≙ [ VideoShop; V
ΞVideoShop ≙ [ ΔVideoShop | θ VideoShop = θ VideoShop
ΔVideoShopHire ≙ (ΞVideoShop \ (hired, hired'))
```

**Figure 9.5** The Formaliser syntax editing tool

The remainder of this chapter focuses upon the application of logic to support the design of CSCW systems. This decision is justified by a number of arguments. Firstly, logic forms a key component of most undergraduate degrees in computing science and engineering. This supports the skill base that is necessary for the pragmatic application of these techniques within commercial development practices. Secondly, there exist a range of relatively simple transformations between other formalisms, such as Petri Nets, and first-order logic. This offers designers the possibility of recruiting different notations during different stages of the development process. Finally, logic programming environments, such as that supported by PROLOG, offer a means of deriving prototype implementations from abstract specifications. As mentioned, this is vital if designers are to validate the products of formal analysis.

## 9.2 STARTING FROM THE GROUND UP: THE APPLICATION OF FORMAL METHODS TO CSCW

The limitations of natural language stem from the fact that it is difficult to write down the exact syntactic rules which guide its use. Similarly, it can be difficult to agree upon the semantics of particular words. Dictionaries provide many different definitions for common words and phrases. Even human factors experts disagree about the meaning of terms such as “workload” [Kan88]. Given such ambiguity and inconsistency, people have long sought to strip away the clutter of everyday language to focus in upon the essentials of communication. Much of this work has built upon the use of mathematics to define syntactic rules for the development of valid sentences. The same mathematical constructs can also be used to specify an exact semantics for these phrases. For example, the designers of a CSCW interface must consider the commands that can be issued by system operators. The following clause might be used to specify that $user\_1$ issues input to quit the application. The intention is to express requirements in a clear and simple manner without the elaborate syntax of natural language:

$$input(user\_1, quit). \tag{9.4}$$

$User\_1$ issues input to quit the system.

Even with such simple beginnings it is possible to reason about the design of a potential interface. For example, the previous clause does not state that other operators, $user\_2$, $user\_3$ etc., also issue input to quit the system. In other words, clause (9.4) does not require agreement between multiple operators. Designers must identify such collaborative requirements if they are to provide the additional cues and prompts that are necessary to achieve coordination. A further benefit is that additional requirements can be gradually introduced as development progresses. For example, the following clause states that the application is shut down if $user\_1$ or $user\_2$ issues input to quit the system:

$$shut\_down(system) \Leftarrow input(user\_1, quit) \vee input(user\_2, quit). \tag{9.5}$$

The application is shut down if user one or user two issues input to quit the system.

This clause relies upon logic operators, $\vee$ (read as "or") and $\Leftarrow$ (read as "if"). These provide the syntax that is needed to construct more complex requirements out of basic relationships such as $input(user\_1, quit)$. We have previously argued that natural language cannot easily be used to support the large-scale development of CSCW systems because it may have ambiguous semantics. We face a similar problem here. What is the meaning of $\vee$ or of $\Leftarrow$? Fortunately, there are a number of techniques that can be used to capture the meaning of such operators. For example, the following truth table provides the semantics for the $\vee$ operator. The first line states that whenever we know that $P$ is true and we know that $Q$ is true then it is safe to conclude that $P \vee Q$ is true. The second line states that whenever we know that $P$ is true and $Q$ is false then it is safe to conclude that $P \vee Q$ is true. The rest of the table can be read in a similar fashion. It is important to emphasize, however, that the formal development of software requires more complex tools than truth tables. The following table is included to reinforce the central idea behind formal specifications. Mathematical structures restrict and focus the components of requirements documents so that they have a precise and concise meaning:

| P | Q | $P \vee Q$ |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

Rather than present a complete introduction to first-order logic, the remainder of this chapter focuses upon the application of formal methods to support the design of CSCW systems. Hodges provides a fuller description of the underlying mathematics [Hod77]. Natural language annotations will be provided in the following pages to help readers who are more interested in the application of the logic rather than its theoretical foundations.

### 9.2.1 An Example Application

We are concerned that a real-world example is used to illustrate our approach. The following pages, therefore, investigate the design of a control room for an oil production facility. These systems have posed a significant challenge for both systems designers and human factors specialists [War89]. Oil production facilities are complex applications. For instance, operators must monitor the extraction of oil from geological structures under the sea-bed. They must also control the extraction and purification of any gas which is recovered with the oil. The UK Government's Gas Conservation Policy prevents these gas products from being "flared" or burnt on the rig. Operators must also monitor repair activities and maintenance schedules. This involves the coordination of many different teams. These properties of the application help to ensure that oil production control systems exhibit many of the problems that frustrate the design of CSCW applications. Groups of operators must monitor computer displays in order to identify faults in many different processes. Users must detect and coordinate their responses to a range of potential errors. Information systems present their operators with information about the extraction of oil products from geological structures deep beneath the sea-bed. Not only must users monitor the rate of extraction but they must also maintain a constant watch for problems that threaten the safety of the rig. For instance, gas leaks pose a considerable risk of fire. If gas is detected then control-room personnel must investigate the cause and identify potential solutions.

## 9.3 DIALOGUE SEQUENCES

First-order logic provides a means of focusing in upon critical properties of interfaces to applications such as the oil production control system. Designers can represent and reason about a design without being forced to consider the low-level details of device polling and event handling. An important point in all of this is that the elements of a specification should provide a common focus for multi-disciplinary design teams. For instance, it might be stated that a fault monitoring system is ready to start logging failures if a user issues a command to start. In order to satisfy such a requirement, interface designers must enable users to easily issue such high-frequency commands. Application engineers must support the functionality that lies behind these commands. A key issue here is that the use of the formal notation does not bias or pre-judge the work of these groups. For instance, the designer is not forced to consider which devices will be used to issue the $start$ command. The choice of presentation strategies can profoundly affect the usability of the final interface. Formal methods can be used to construct a design without forcing commitment to a particular implementation early in the development process:

$$\begin{aligned} &start\_logging \Leftarrow \\ &\quad input(user\_1, start) \wedge effect(start, off, logging). \end{aligned} \tag{9.6}$$

The monitoring system starts logging faults if $user\_1$ issues input to start the application and the effect of that input is to transform the state of the system from one in which it is off to one in which it is logging faults.

First-order logic can be recruited to reason about the complexity of concurrent interaction between multiple users. Contention is a frequent problem in multi-user systems which allow two or more operators to access the same resources. For example, one user might attempt to quit the application while another attempts to log a fault:

$$\begin{aligned} &log\_contention \Leftarrow \\ &\quad input(user\_1, quit) \wedge input(user\_2, log\_pump\_A\_error). \end{aligned} \tag{9.7}$$

Contention arises in the logging system if $user\_1$ issues input to quit the system and $user\_2$ issues input to log a fault.

This conflict could be resolved by always giving priority to commands from a particular user [Pen90]. Alternatively, priority might be associated with certain commands [Ell91]. Input with a lower priority may be disregarded. The input $quit$ does not affect the state of the system:

$$\begin{aligned} &resolve\_contention \Leftarrow log\_contention \wedge \\ &\quad effect(quit, on, on) \wedge effect(log\_pump\_A\_error, on, pump\_A\_error). \end{aligned} \tag{9.8}$$

Contention is resolved if the logging command takes effect but input to quit the system does not change the state of the application.

Unfortunately, a number of problems must be resolved before first-order logic can be used to support the design of concurrent multi-user systems. In particular, there is no notion of ordering in first-order logic. This creates problems because many critical issues in the development of CSCW systems arise from the sequencing of events. In our example, no conflict need arise if the system were closed down after the fault had been recorded. As there is no notion of sequence in first-order logic, the previous clause would still specify that contention occurs even if $quit$ were issued some time after $log\_pump\_A\_error$. Temporal sequencing must be introduced if such concurrency requirements are to be made explicit within logic specifications of interactive systems.

## 9.3.1 Time and First-Order Logic

The lack of sequencing in first-order logic has important consequences for the design of CSCW systems. Delays in receiving information, from systems and other users, can lead to breakdown and referential failure [McC91]. Concurrent input can lead to contention and interference. The following section describes how the temporal properties of an interface can be made explicit within logic specifications. This provides the designer with a medium in which to reason about the possible impact of timing properties upon the users of CSCW applications.

### 9.3.1.1 Fixed Time-Stamps

Fixed time-stamps provide one means of avoiding the limitations of first-order logic. This approach associates a particular instant of time with each clause in a specification. For example, it might be specified that $quit$ and $log\_pump\_A\_error$ should be input at twenty seconds past midday. An additional requirement might also be that the command to quit the system should not take effect when the fault is being logged at twenty-five seconds past midday. An important point here is that time-stamps help to build a standard time-line for critical requirements. This provides a means of explicitly representing synchronization requirements:

$$\begin{aligned}&fixed\_solution \Leftarrow \\ &\quad log\_contention(120020) \wedge effect(quit, on, on, 120025) \wedge \\ &\quad effect(log\_pump\_A\_error, on, pump\_A\_error, 120025).\end{aligned} \tag{9.9}$$

Contention is resolved if $quit$ and $log\_pump\_A\_error$ are input at twenty seconds past midday and five seconds later the monitoring system logs the fault.

There are a number of limitations which restrict the utility of fixed time-stamps within a specification. Considerable burdens are imposed upon the designer who must provide and maintain the temporal parameters in each clause. A further problem is that it is difficult to represent persistent properties of CSCW interfaces. For instance, a designer might wish to ensure that $quit$ does not take effect before $log\_pump\_A\_error$:

$$\begin{aligned}&persistent\_solution \Leftarrow log\_contention(120020) \wedge \\ &\quad not(effect(quit, on, off, 120021)) \wedge not(effect(quit, on, off, 120022)) \wedge \\ &\quad not(effect(quit, on, off, 120023)) \wedge not(effect(quit, on, off, 120024)) \wedge \\ &\quad effect(log\_pump\_A\_error, on, pump\_A\_error, 120025).\end{aligned} \tag{9.10}$$

Contention is resolved if $quit$ and $log\_pump\_A\_error$ are input at twenty seconds past midday and the input to quit the system does not take effect at twenty-one seconds past midday, twenty-two seconds past midday, twenty-three seconds past midday, twenty-four seconds past midday and the monitoring system logs the fault at twenty-five seconds past midday.

Fixed time-stamps also introduce a high degree of temporal determinism into a specification. In order to fulfill the previous specification both users must provide concurrent input at exactly twenty seconds past midday. If designers wished to represent means of avoiding contention at twenty seconds before midday, at twenty seconds to one, at half past four or at any other time, they would be forced to repeat previous clauses for each of these points.

### 9.3.1.2 Time Variables

The limitations of fixed time-stamps can be avoided by using time variables. For example, $fixed\_solution$ (9.9) might be re-expressed as follows:

$$\begin{aligned}&variable\_solution \Leftarrow log\_contention(T) \wedge effect(quit, on, on, T1) \wedge \\ &\quad effect(log\_pump\_A\_error, on, pump\_A\_error, T1) \wedge after(T, T1).\end{aligned} \tag{9.11}$$

Contention is resolved if $user\_1$ and $user\_2$ issue input at time $T$ and the command to $quit$ the system is ineffective at some subsequent time, $T1$, when $user\_2$'s fault is logged.

The time variables, $T$ and $T1$, could be instantiated at a number of points during interaction and the temporal ordering is made explicit by the predicate *after*. Unfortunately, the use of such variables still imposes considerable burdens upon the interface designer. It is particularly important that a clear semantics is maintained for predicates, such as *after*, which define an ordering over variables. These can radically effect the properties of any specification. For example, the following clause specifies that $user\_1$'s input does take effect after the fault has been logged:

$$\begin{aligned} &circular\_solution \Leftarrow \\ &\quad log\_contention(T) \wedge effect(quit, on, on, T1) \wedge \\ &\quad effect(log\_pump\_A\_error, on, pump\_A\_error, T1) \wedge \\ &\quad effect(quit, pump\_A\_error, off, T2) \wedge after(T, T1) \\ &\quad \wedge after(T1, T2) \wedge after(T2, T). \end{aligned} \tag{9.12}$$

Contention is resolved if $user\_1$ and $user\_2$ issue input at time $T$ and the command to $quit$ the system is ineffective at some subsequent time, $T1$, when $user\_2$'s fault is logged but the input to $quit$ the system does take effect at time $T2$.

The previous clause illustrates some of the problems that can arise in large-scale specifications of CSCW systems. In particular, time $T2$ occurs both after and before time $T$. This circular model of time makes little sense. Unfortunately, there is a high risk of such considerable problems occurring if designers are forced to construct complex sequences in terms of the *after* relation. Temporal ambiguities may easily occur in specifications that contain hundreds or thousands of clauses, especially if they must be constructed and maintained by many different development teams.

### 9.3.1.3 Temporal Logic

Temporal logic extends first-order logic by supporting the following operators: $\Diamond$ (read as "eventually"); $\bigcirc$ (read as "next"); $\Box$ (read as "always") and $\mathcal{U}$ (read as "until") [Man81]. This notation relieves the designer from the burdens of maintaining an explicit ordering in terms of predicates such as *after*. The ordering is captured within the definition of temporal operators. For example, $\Diamond$ may be defined using a set of time-stamps $T$, $|w|_t$ denotes the truth value of the formula $w$ at time $t$. It is important to note, however, that designers can simply introduce temporal operators into a specification. They are not forced to explicitly represent the *after* sequences that are embedded within the definitions of temporal operators. Nor are they obliged to explicitly deal with the underlying model represented in the following definition:

$$|\Diamond(w)|_t \equiv \exists t1 \in T[after(t, t1) \wedge |w|_{t1}] \tag{9.13}$$

The $\Diamond$ operator is defined such that any formula $w$ is eventually true at time $t$ if there exists some later time, $t1$, when $w$ is true.

Prior provides complete definitions for the various temporal operators mentioned above [Pri67]. In contrast, our focus is upon the application of the notation. The following section, therefore, shows how temporal logic can be used to analyse solutions for the problem of interference within our CSCW application.

#### 9.3.1.4 Input Priorities Revisited

Contention can be resolved by associating priorities with commands. Scarce resources can be allocated to input with a high priority, input with a low priority may be disregarded. In terms of our oil production system, a command to switch off the fault monitoring application might be assigned a relatively low priority. The systems should continue to log faults whenever possible and input to disable the system might, therefore, be ignored if users continue to report problems in their equipment. Unfortunately, this solution suffers from a number of limitations. There is no guarantee of fairness, some users may be "frozen" out of interaction if their commands always receive low priority. In particular, a user could not predict the success or failure of a quit command unless they could determine the priority of concurrent input from all other users. A designer might reduce this uncertainty by ensuring that low-priority input is eventually effective:

$$\begin{aligned} &priority\_solution \Leftarrow log\_contention \wedge \\ &\quad effect(log\_pump\_A\_error, on, pump\_A\_error) \wedge \\ &\quad \Diamond\, effect(quit, pump\_A\_error, off). \end{aligned} \tag{9.14}$$

Contention is reduced if input to log a fault takes effect in the present interval and eventually the input to quit the system takes effect.

This approach can be used to develop sophisticated priority structures. For example, a command to close the system might be assigned a lower priority than input to log a fault in the emergency deluge equipment for fire-fighting on the rig. This, in turn, might be assigned a higher priority than the input to log a pump fault. The following clause formalizes this requirement. It is clearly important to explicitly represent these priorities if critical input is not to be delayed:

$$\begin{aligned} &ranking\_solution \Leftarrow input(user\_1, quit) \wedge \\ &\quad input(user\_2, log\_pump\_A\_error) \wedge \\ &\quad input(user\_3, log\_deluge\_failure) \wedge \\ &\quad effect(log\_deluge\_failure, on, deluge\_failure) \wedge \\ &\quad \Diamond(effect(log\_pump\_A\_error, deluge\_failure, fire\_risk\_alert) \wedge \\ &\quad \Diamond\, effect(quit, on, off)). \end{aligned} \tag{9.15}$$

Contention is reduced if three users issue input at the same time to close down the system, to log a pump fault and to log a fault in the emergency deluge system. The input to log the deluge fault takes effect immediately and eventually the pump failure is recorded. This changes the state of the system into one in which there is a fire risk and eventually at some point after this the input to close down the system will have the effect of turning the system off, providing the state has returned to normal.

Unfortunately, postponing the effect of low-priority input can cause a number of problems for the users of groupware applications. The previous clause does not specify when the $\diamond$ (read as "eventually") clause will be true. Delays in system responses can lead to frustration and error [Kuh89]. Unpredictable behavior is likely to occur when periods of quiescence allow the system to process a backlog of low-priority input [Ell89]. Delayed commands might take effect at inappropriate moments during an interaction. The presentation of a large amount of contextual information is required before a user can resolve such instances of unpredictability.

### 9.3.1.5 Locking

Interference can occur even if input priority mechanisms are adopted. Low-priority input to halt the system might take effect before another user has finished logging a fault. This interference can be avoided by assigning priorities to transactions rather than single commands. For example, transaction locking restricts input from other users until an operation has been terminated. Input priority, user priority or first-come first-served mechanisms provide a means of determining the identity of the next user to "gain the floor":

$$\begin{aligned} transaction\_lock \Leftarrow {} & input(user\_2, log\_pump\_A\_error) \wedge \\ & (not(input(user\_1, I)) \ \mathcal{U} \ \ input(user\_2, end\_pump\_A\_error)). \end{aligned} \tag{9.16}$$

Contention is reduced through the imposition of a lock if $user\_1$ cannot enter any input, $I$, until $user\_2$ has cleared the fault.

Unfortunately, single-entry transaction locking resolves contention by restricting multi-user systems to sequential interaction. There are a number of reasons why such an approach is often unacceptable. Users may not relinquish control if transactions are not terminated. Opportunism and negotiation may provide more fruitful grounds for cooperation than prescription. In contrast to transaction locking, data locking avoids contention by restricting the ability of operators to make modifications to shared resources. For instance, $user\_1$ might continue to interact with the fault monitoring system even though $user\_2$ is logging a fault on $pump\_A$. Designers may only choose to prevent $user\_1$ from also logging a fault on that component while $user\_2$ is accessing it:

$$\begin{aligned} logging\_lock \Leftarrow {} & input(user\_2, log\_pump\_A\_error) \wedge \\ & (not(input(user\_1, log\_pump\_A\_error))\mathcal{U} \\ & input(user\_2, end\_pump\_A\_error)). \end{aligned} \tag{9.17}$$

Contention is reduced if $user\_1$ cannot log a fault until $user\_2$ has finished logging their fault.

It is important to notice that this solution has been expressed without reference to device primitives or particular polling strategies. Later sections will describe tools which have been developed to directly execute such abstract specifications. This provides a means of evaluating the consequences of placing restrictive locks upon the group process. For example, this approach can prove unnecessarily restrictive if locks are placed upon entire systems. Interference need not occur if users make concurrent updates to different processes. Alternatively, as we have seen, data locks may be imposed at the level of individual systems or sub-components. This introduces considerable complexity into the design of an interface [Gre87]. For instance, logging knock-on faults can involve the acquisition of a large number of locks. The process by which a user requests and relinquishes a shared resource can impose a large overhead on the times necessary to perform even simple operations.

## 9.4 FORMALIZING THE PRESENTATION OF CSCW SYSTEMS

The second way in which formal methods can be applied to support CSCW systems is in display design. This poses significant challenges because the presentation of multi-user applications is qualitatively different from that of single-user systems. Some displays are shared amongst the members of a group while others are not. For example, the task of monitoring oil production will require different information from that of gas extraction. This, in turn, will require different information from the task of fire prevention and detection. CSCW designers must consider the composition of displays that support these different activities. This development problem is complicated because the individual elements of a display will change over time. It is critical that development teams have some means of representing and reasoning about these common and private contexts if they are to provide adequate support for group activities and individual tasks.

### 9.4.1 Unstructured Graphics

Unstructured graphical representations do not distinguish between the images of display components, such as menus and icons. For instance, bitmaps represent the image of pixels as bits in a data structure. Designers might use these representations to specify the images that are presented to the multiple users of CSCW systems, such as the oil production application:

```
DeclareBitmap(logging_display.bit, 42, 49, logging_display.bits);
short on_display.bits[] =
/* Abbreviated for the sake of brevity */
{
  0x0000, 0x0000, 0x0000, 0x000f, 0xff00, 0x0000, 0x007f, 0xffc0,
  0x0000, 0x00ff, 0xfff0, 0x0000, 0x00ff, 0xfff0, 0x0000, 0x00ff,
  0x001f, 0x0000, 0x0340, 0x006f, 0x0000, 0x03b0, 0x0a97, 0x0000,
  0x037d, 0x3fef, 0x0000, 0x03ee, 0x0a1b, 0x0000, 0x03d7, 0x3ff7,
  0x0000, 0x03fd, 0x87ca, 0x0000, 0x03f7, 0x5616, 0x0000, 0x014b,
  0x0000, 0x0000, 0x0000,
};
```

It is extremely difficult to decompose data structures, such as the previous bitmap, into the components of a complex image. This hinders the development of multi-user computer systems because, typically, only part of a screen is shared by all system operators. The common parts of a display cannot easily be extracted from an unstructured representation.

### 9.4.2 Procedural Graphics

Procedural graphics systems construct pictures from sequences of instructions. Designers might use these systems to generate interface components without describing the entire appearance of a display. The shared images of CSCW systems can be represented and reasoned about in terms of the instructions necessary to create them. For instance, the following clauses show how the $\bigcirc$ (read as “next”) operator can be used to describe the instructions that are necessary to draw part of a $pump\_A\_error\_icon$:

$$\begin{array}{l} draw\_pump\_A\_error\_icon \Leftarrow \\ \quad pen\_down \wedge \bigcirc(pen\_forward(10) \wedge \\ \quad \bigcirc(pen\_rotate(90) \wedge \bigcirc(pen\_forward(20) \wedge \\ \quad \bigcirc(pen\_rotate(90) \wedge \bigcirc(pen\_forward(10) \wedge ...)))) \end{array} \tag{9.18}$$

The error icon for $pump\_A$ is drawn if in the present interval the pen is lowered to the paper and in the next interval the pen is moved forward by ten units and in the next again interval the pen is rotated by ninety degrees and...

Procedural approaches offer only limited support for the prototyping of multi-user CSCW systems. Designers would be forced to write many thousands of instructions in order to create complex images. This burden is greatly increased because different sequences of instructions must simultaneously be executed on a range of different devices in order to present displays to a number of different users. If one instruction were omitted or placed out of sequence then the final image might be corrupted. Szekely and Myers identify a further limitation of procedural graphics systems [Sze88]. If users select part of a display, using a mouse or some cursor keys, then there is no means of identifying the target of their selection using the instructions that generated the image. Designers must, therefore, maintain additional data structures in order to determine which images are selected by operator input. This is a considerable overhead for prototype CSCW systems whose users may concurrently select many different parts of many different images.

### 9.4.3 Structured Graphics

Logic can be used to represent the images presented by a CSCW system at an extremely high level of abstraction. For instance, the following clause specifies that $user\_1$ is presented with a $condensate\_display$, $user\_2$ is presented with a $deluge\_display$. Similar clauses might be introduced to represent the images presented to $user\_3$, $user\_4$, $user\_5$ etc:

$$display(user\_1, condensate\_display). \tag{9.19}$$
$$display(user\_2, deluge\_display). \tag{9.20}$$

The first clause states that $user\_1$ is presented with the $condensate\_display$. The second clause states that $user\_2$ is presented with the $deluge\_display$.

Display abstractions can be decomposed into their component parts. For instance, the $condensate\_display$ presented to $user\_1$ might show that the pneumatic valves, the centrifuges and the non-return valves are all functioning correctly but that there is an error with pump A. This image is illustrated in Figure 9.6. The structure of the $user\_1$’s $condensate\_display$ is represented by the following clauses:

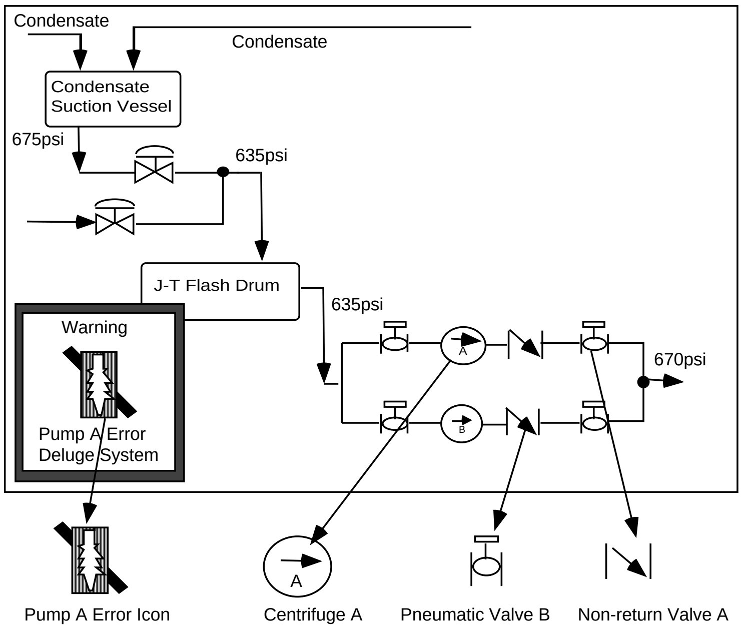


**Figure 9.6** The graphical decomposition of the *condensate_display*

$$part(user\_1, condensate\_display, centrifuge\_A). \tag{9.21}$$
$$part(user\_1, condensate\_display, pneumatic\_valve\_B). \tag{9.22}$$
$$part(user\_1, condensate\_display, pump\_A\_error\_icon). \tag{9.23}$$
$$part(user\_1, condensate\_display, non\_return\_valve\_A). \tag{9.24}$$

The first clause states that the $centrifuge\_A$ icon is part of the $condensate\_display$ presented to $user\_1$. The second clause states that the $pneumatic\_valve\_B$ icon is part of the $condensate\_display$ presented to $user\_1$. The third clause states that the $pump\_A\_error$ icon is part of the $condensate\_display$ presented to $user\_1$ and so on.

Figure 9.7 illustrates how the $deluge\_display$ presented to $user\_2$ can be decomposed in a similar fashion. The structure of this image can be represented by the following clauses:

$$part(user\_2, deluge\_display, pump\_A\_error\_icon). \tag{9.25}$$
$$part(user\_2, deluge\_display, inlet\_B\_capacity). \tag{9.26}$$
$$part(user\_2, deluge\_display, pump\_C\_icon). \tag{9.27}$$
$$part(user\_2, deluge\_display, protection\_cage\_C). \tag{9.28}$$

The first clause states that the $pump\_A\_error$ icon is part of the $deluge\_display$ presented to $user\_2$. The second clause states that the $inlet\_B\_capacity$ is part of the $deluge\_display$ presented to $user\_2$. The third clause states that the $pump\_C\_icon$ icon is part of the $deluge\_display$ presented to $user\_2$ and so on.

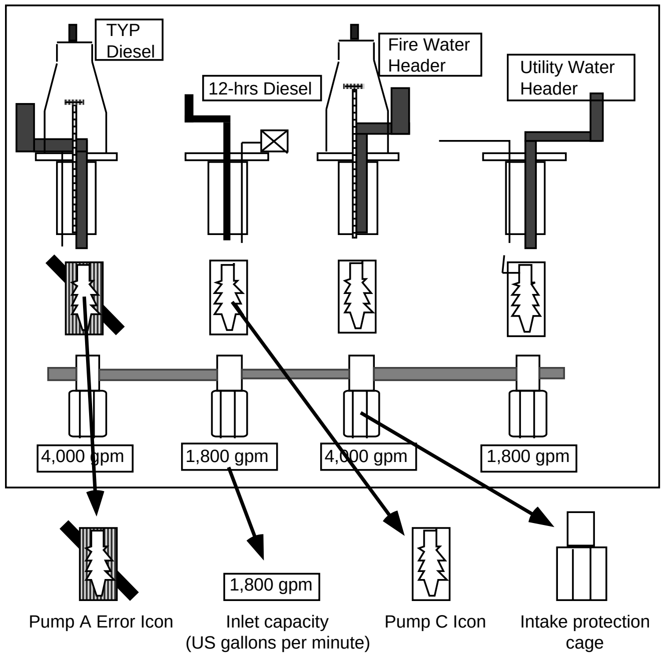


**Figure 9.7** The graphical decomposition of the *deluge_display*

Designers can use logic clauses to identify those images, such as $pump\_A\_error\_icon$, which form the common context of operations performed by $user\_1$ and $user\_2$. This supports the detailed development of CSCW systems. For instance, designers might specify that the deluge pumping equipment is closed if $user\_1$ and $user\_2$ are presented with an error for pump A and both provide input to close off the pump. Such an agreement would be appropriate because closing-off fire-safety equipment has important consequences for the oil and gas extraction processes. The display requirement that they both see the error icon for $pump\_A$ is intended to ensure that both operators are presented with sufficient contextual information in order for them to coordinate their response:

$$\begin{aligned} & voting\_close\_pump\_A \Leftarrow \\ & \quad display(user\_1, condensate\_display) \wedge \\ & \quad display(user\_2, deluge\_display) \wedge \\ & \quad part(user\_1, condensate\_display, pump\_A\_error\_icon) \wedge \\ & \quad part(user\_2, deluge\_display, pump\_A\_error\_icon) \wedge \\ & \quad \Diamond(input(user\_1, close\_pump\_A) \wedge \\ & \quad input(user\_2, close\_pump\_A)). \end{aligned} \tag{9.29}$$

This states that a voting system is used to close $pump\_A$ if $user\_1$ is presented with the $condensate\_display$ and $user\_2$ is presented with the $deluge\_display$ and $pump\_A\_error$ icon is part of both displays and eventually both $user\_1$ and $user\_2$ provide input to close the $pump$.

Such clauses support further stages in the development of CSCW systems. For instance, it has not been specified that the $deluge\_display$ presents detailed information about the centrifuges that are used during gas extraction from the oil. It would not, therefore, be appropriate to expect $user\_2$ to resolve problems with these components without access to additional data. Contention might occur if they did attempt to operate a centrifuge.

For example, $user\_1$ might $close$ it while $user\_1$ tried to $open$ it. The display abstractions introduced in the previous paragraphs might be integrated with the temporal operators from the first part of this chapter to specify solutions for such problems. Designers might require that a lock is imposed to resolve contention if $user\_2$ is not presented with information about a particular centrifuge:

$$\begin{aligned} & lock\_out\_centrifuge\_contention \Leftarrow \\ & \quad input(user\_1, close\_centrifuge\_A) \wedge \\ & \quad display(user\_2, deluge\_display) \wedge \\ & \quad not(part(user\_2, deluge\_display, centrifuge\_A)) \wedge \\ & \quad not(input(user\_2, I)\ \mathcal{U}\ \ input(user\_1, end\_centrifuge\_A\_error)). \end{aligned} \tag{9.30}$$

This states that a lock is imposed to prevent contention over $centrifuge\_A$ if $user\_1$ provides input to close it and $user\_1$ is presented with the $deluge\_display$ and the $centrifuge\_A$ icon is not part of that display and $user\_2$ does not provide input until $user\_1$ has issued input to state that the error in the centrifuge is over.

We have argued that problems such as contention and deadlock make it necessary to consider the "look and feel" of a potential interface during the early stages of CSCW systems development. It is, therefore, important that designers can refine high-level clauses, such as $condensate\_display$, into the primitive images which are presented to users. One means of doing this is to describe images in terms of lines:

$$line(user\_1, centrifuge\_A, 0.1, 0.2, 0.6, 0.2). \tag{9.31}$$

This states that the image of the $centrifuge$ icon presented to $user\_1$ includes a line from coordinates (0.1,0.2) to (0.6, 0.2).

A limitation with this approach is that operator input is not usually directed towards lines but towards areas of the screen. A user selecting an icon does not, necessarily, expect to select a particular line of its image. In order to support such interaction, designers must exploit more sophisticated graphical "building-blocks". Figure 9.8 illustrates how the image of the *condensate_display* can be described in terms of a number of regions: a background region, a text region and a centrifuge region. Regions can be further decomposed into sub-regions. Each region has properties, such as size and position, attributes, such as font and pattern, and a behavior, such as whether or not it is selectable. For instance, the $centrifuge\_A$ icon could be presented to $user\_1$ as a region with a blank background and dimensions that occupy one-twentieth of the screen:

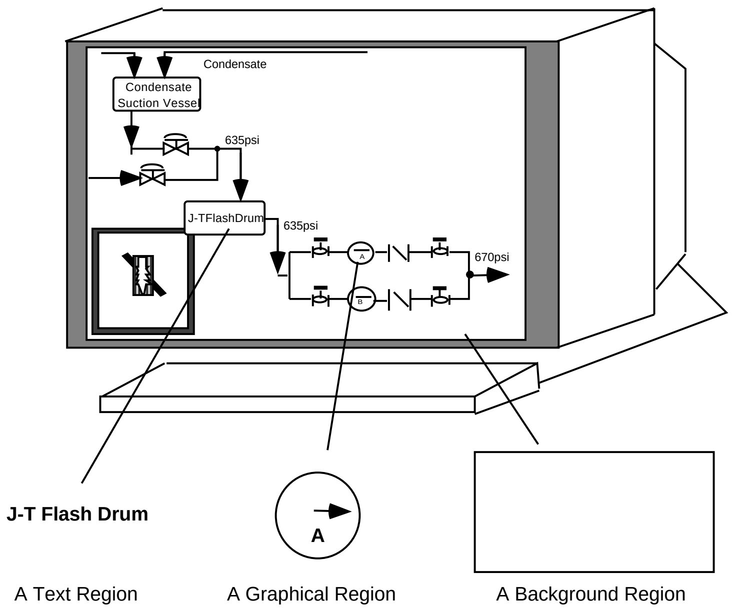


**Figure 9.8** The region decomposition for part of the *condensate_display*

$$dimension(user\_1, centrifuge\_A, 0.05, 0.05). \tag{9.32}$$
$$pattern(user\_1, centrifuge\_A, blank). \tag{9.33}$$

The first clause states that the image of the $centrifuge\_A$ icon presented to $user\_1$ has dimension that occupy one twentieth of the screen. The second clause states that the image of the $centrifuge\_A$ icon presented to $user\_1$ has a blank background.

These clauses can represent the ways in which CSCW displays must be tailored in order to support group tasks. For instance, designers might require that $user\_2$ can monitor the effects of $user\_1$'s intervention on the $centrifuge$ while performing other duties. Under such circumstances, the centrifuge might be introduced into the $deluge\_display$. The dimensions of the centrifuge icon could also be reduced in order to free display resources for the presentation of $user\_2$'s primary activities. Although both users must be presented with information about the centrifuge, the size of this image is used to reflect the relative importance of the component for each user's task. The following clause illustrates how logic abstractions can be used to represent and reason about CSCW systems which support semi-independent views [Ell91] of application processes :

$$part(user\_2, deluge\_display, centrifuge\_A). \tag{9.34}$$
$$dimension(user\_2, centrifuge\_A, 0.02, 0.02). \tag{9.35}$$

The first clause states that the $centrifuge\_A$ icon is part of the $deluge\_display$ presented to $user\_2$. The second clause states that the image of the icon presented to $user\_2$ has dimension that occupy one-fiftieth of the screen.

The choice of input media has a profound affect upon the usability of CSCW systems. For instance, Galer and Yap have used prototypes to investigate the costs and benefits of different input devices for the users of intensive care systems [Gal80]. Some operators suffered from high error rates when using thumb wheels, mice were difficult to use in cluttered clinical environments. In order to evaluate the tradeoffs that exist between tracker-balls, mice, joysticks and keyboards, the designers of CSCW systems must be able to represent a variety of input devices.

### 9.4.4 Introducing Input Information

Input can be represented by introducing device drivers into formal specifications. For instance, the following routine "blinks" the caret when a mouse is moved over a text region in an Apple Macintosh [App86]:

```
CLR.L          -SP       ;event code for null event is 0
PEA            2(SP)     ;pass null event
CLR.L          -SP       ;pass NIL dialogue pointer
CLR.L          -SP       ;pass NIL pointer
DialogueSelect           ;invoke DialogueSelect
ADDQ.L         #4,SP     ;pop off result and null event
```

Burton et al show how designers might formalize similar code in order to specify single-user graphical interfaces built from the Apple Macintosh Toolbox [Bur89]. Such descriptions provide an appropriate level of detail for many stages in development. They are, however, extremely device dependent. The complexity of accessing input at this level of detail might dissuade designers from assessing the costs and benefits of a range of devices for the many different users of CSCW systems. Like bitmaps, this approach provides a one-step refinement between abstract, formal representations of graphical interfaces and device specific implementations. This would have important consequences for the development of embedded control systems, such as our oil rig application. In these environments, CSCW applications must frequently be developed to run on a range of existing hardware. It would not be acceptable to rebuild a control room because its input devices could not be formalized in terms of their device drivers.

Input from a range of physical devices, such as mice or tracker balls, can be represented by generic events, such as $on\_select$ and $on\_move$. Events can be introduced into formal specifications by associating them with graphical regions. For example, the following clause shows how designers might specify that $pump\_A$ is closed if $user\_1$ and $user\_2$ are presented with an $pump\_A\_error\_icon$ and both operators use a mouse to select this image. This clause can also be used to describe control rooms in which the operators had access to tracker-balls or cursor keys instead of mice. These devices could also generate $on\_select$ events. Such device independence helps to avoid premature commitment to particular hardware platforms. Implementation decisions can be postponed until late in the development cycle when the costs and benefits of a range of different input media have been considered. This encourages designers to identify those devices that are most appropriate to the particular tasks and environments of CSCW groups:

$$\begin{aligned}
& event\_close\_pump\_A \Leftarrow \\
& \quad display(user\_1, condensate\_display) \wedge \\
& \quad display(user\_2, deluge\_display) \wedge \\
& \quad part(user\_1, condensate\_display, pump\_A\_error\_icon) \wedge \\
& \quad part(user\_2, deluge\_display, pump\_A\_error\_icon) \wedge \\
& \quad \Diamond(input(user\_1, pump\_A\_error\_icon, on\_select) \wedge \\
& \quad (not(effect(on\_select, pump\_A\_error\_icon, pump\_A\_off)\mathcal{U} \\
& \quad input(user\_2, pump\_A\_error\_icon, on\_select)). \qquad (9.36)
\end{aligned}$$

This states that input events are used to close $pump\_A$ if $user\_1$ is presented with the $condensate\_display$ and $user\_2$ is presented with the $deluge\_display$ and the $pump\_A\_error$ icon is part of the fault and line displays and eventually $user\_1$ issues a select event for the icon but this is not effective until $user\_2$ also issues a select event on the icon.

We have shown that formal notations can be used to represent the proportion and location of graphical images on a display. A limitation with this approach is that it does not consider the operators' physical and environmental surroundings. Specifying the size and position of an image is of little benefit if users cannot easily view the devices that are used to present the $condensate$ and $deluge$ displays. This is a weakness of almost all previous approaches to interface design. Few existing techniques consider the layout of particular working environments.

## 9.5 WORKING ENVIRONMENTS

The European Community Directive on work with Display Screen Equipment and the United Kingdom's Health and Safety Regulations provide guidelines on the correct layout of working environments for computer operators. Screens should be parallel to overhead fluorescent tubes, at right angles to windows etc. Unfortunately, many techniques in human–computer interaction completely ignore these issues. They provide ample support for screen layout and dialogue design but they provide no means of reasoning about the physical layout of work environments. Conversely, the empirical techniques and CAD tools that have been developed to analyse different operator postures do not address the concerns that dominate human–computer interaction [Mal89]. The lack of integration between user-interface design and environmental layout is not a serious problem in many contexts. Office workers can easily move keyboards, screens and telephones into positions that support their everyday tasks. This lack of integration is, however, a more serious problem for the development of safety-critical applications. The position of a display can determine whether operators will observe a warning within a particular time period [Wic84]. The physical location of buttons, keyboards and mice can affect the error rates for particular input sequences [Joh94b]. For example, the following clause states that $user\_1$ is responsible for observing and responding to the failure of a blow-back valve. These devices ensure that material is not forced back up a line from which it is being pumped:

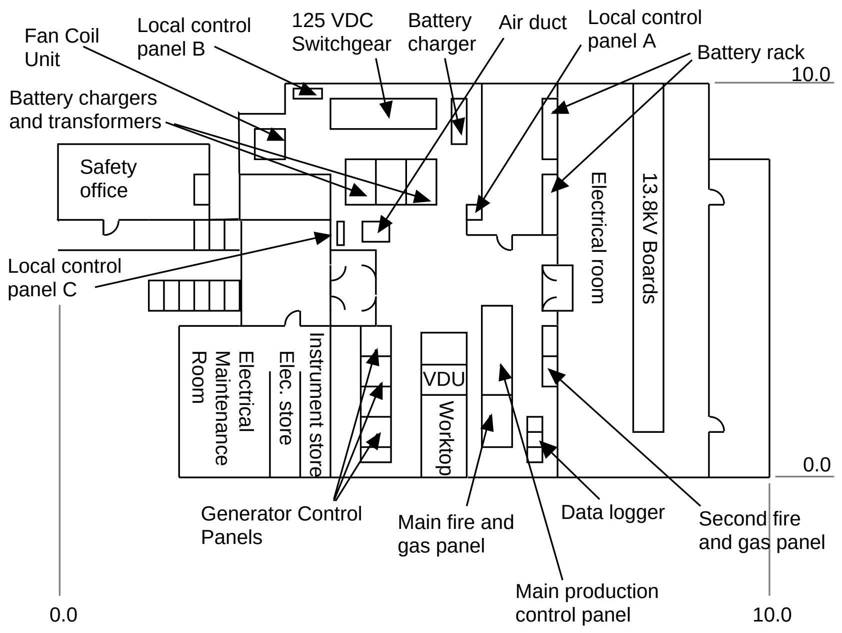


**Figure 9.9** Control room module for North Sea oil production

$$\begin{aligned}
&user\_1\_responsible\_for\_closing\_valve\_A \Leftarrow \\
&\quad display(user\_1, condensate\_display) \wedge \\
&\quad display(user\_2, deluge\_display) \wedge \\
&\quad part(user\_1, condensate\_display, valve\_A\_error\_icon) \wedge \\
&\quad not(part(user\_2, deluge\_display, valve\_A\_error\_icon)) \wedge \\
&\quad input(user\_1, valve\_A\_error\_icon, on\_select). \qquad (9.37)
\end{aligned}$$

This states that users agree to close $valve\_A$ if $user\_1$ is presented with the $condensate\_display$ and $user\_2$ is presented with the $deluge\_display$ and $valve\_A\_error\_icon$ is part of the condensate and but not of the deluge display and $user\_1$ provides input to close $valve\_A$.

Such dialogue requirements make implicit assumptions about the layout of a potential control room. Designers must ensure that $user\_1$ can view the $valve\_A\_error\_icon$ from their normal working position. Figure 9.9 illustrates that this may be a non-trivial problem. For instance, if the operator were routinely stationed behind the work surface at the bottom on the figure then it would be difficult for them to view a warning presented on the local control panels towards the top of the layout. Fortunately, logic abstractions can also be used to reason about the physical organization of complex working environments.

## 9.6 REPRESENTING WORKSTATION LAYOUT

Designers can exploit logic to represent the allocation of displays to the control panels that users must operate. For instance, clause (9.37) required that $user\_1$ should be presented with the $condensate\_display$. This could be presented through the local control panel next to the switchgear shown in Figure 9.9 rather than through the main VDU next to the worktop:

$$present(user\_1, condensate\_display, local\_panel\_A). \tag{9.38}$$

$$location(local\_panel\_A, 6.0, 6.5). \tag{9.39}$$

$$dimension(local\_panel\_A, 1.5, 0.9, 1.1). \tag{9.40}$$

The first clause states that the condensate display is presented to $user\_1$ through local control panel A. The remaining clauses state that the control panel is located at Cartesian coordinates (6.0, 6.5) and is 1.5 meters in dimension along the X axis, 0.9 along the Y axis and 1.1 meters along the Z axis; this corresponds to the height of the panel.

Designers can use these clauses to guide the detailed layout of a control system. By introducing positional information into logic clauses it is possible to represent the likely working position of an operator performing a particular task. For instance, $user\_1$'s normal activity might be to coordinate the operation of the system from behind the worktop. This would place the user at a position close to $(6.0, 2.2)$. It would then be difficult for $user\_1$ to respond to warnings presented on local control panel A at the same time as monitoring a display on the fire and gas panel:

$$\begin{aligned}
&divided\_attention \Leftarrow \\
&\quad location(user\_1, 6.0, 2.2) \wedge \\
&\quad location(local\_panel\_A, 6.0, 6.5) \wedge \\
&\quad location(fire\_panel, 6.5, 1.5) \wedge \\
&\quad present(user\_1, Display\_1, local\_panel\_A) \wedge \\
&\quad present(user\_1, Display\_2, fire\_panel) \wedge \\
&\quad part(user\_1, Display\_1, valve\_A\_error\_icon) \wedge \\
&\quad part(user\_1, Display\_2, communications\_error) \wedge \\
&\quad input(user\_1, valve\_A\_error\_icon, on\_select) \wedge \\
&\quad input(user\_1, self\_test\_communications, on\_select).
\end{aligned} \tag{9.41}$$

This states that $user\_1$ must divide their attention if they are at (6.0,2.2) and must monitor two different displays, one presented by local panel A at (6.0,6.5) and the other presented by the fire and gas console at (6.5,1.5). And that those displays contain warnings about a communications error and a fault with valve A and $user\_1$ must provide input to resolve those warnings.

Logic can be used to represent potential solutions to such problems. For instance, the position of the fire and gas console might be moved so that it could more easily be observed while $user\_1$ was monitoring the local control panel. This can be represented by altering one

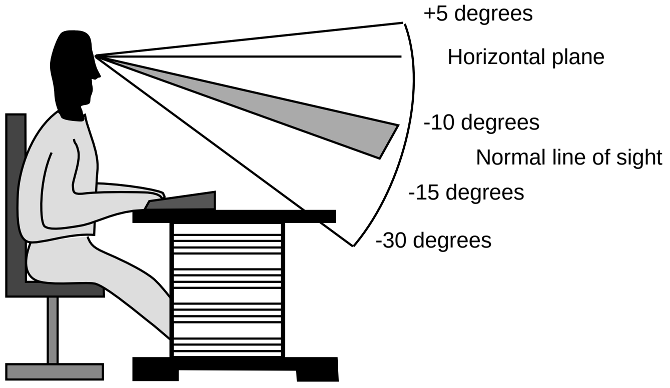


**Figure 9.10** The relaxed viewing angle

of the $location$ clauses. Alternatively, the task of monitoring and responding to the communications error might be allocated to another user. These two potential solutions again illustrate the close interaction between dialogue design and the layout of control rooms:

$$\begin{aligned}
& coordinated\_response \Leftarrow \\
& \quad part(user\_1, Display\_1, valve\_A\_error\_icon) \wedge \\
& \quad part(user\_2, Display\_2, communications\_error) \wedge \\
& \quad input(user\_1, valve\_A\_error\_icon, on\_select) \wedge \\
& \quad input(user\_2, self\_test\_communications, on\_select). \qquad (9.42)
\end{aligned}$$

This states that there is a coordinated response if $user\_1$'s display contains a warning about a fault with valve A and $user\_2$'s display contains a warning about a communications problem and $user\_1$ must provide input to resolve the valve problem and $user\_2$ must resolve the communications error.

Such clauses illustrate the benefits of formal methods for the integration of interface design and environmental layout. It is not clear how the individual images shown to many different operators might be represented using the conventional sketches and two-dimensional plans of control rooms, such as that shown in Figure 9.9.

## 9.7 USING ERGONOMIC GUIDELINES TO INFORM CSCW DESIGN

Research in the field of human factors and ergonomics has developed a mass of information about suitable operator postures and working positions. For instance, Figure 9.10 illustrates Grandjean's [Gra88] guidelines for a relaxed viewing angle from an upright, seated posture. If operators are required to monitor displays outside of the $-10$ to $-15$ degree cone for long periods then static muscle overloading may occur. Until now, it has been difficult to envisage how such information can be used to *directly* inform the development of CSCW systems. In contrast, the previous clauses can be used to reason about the consequences of such figures for interactive dialogues in particular working environments. For example, assuming that $user\_1$ were at the worktop in the centre of the control room at (6.0,2.2,1.3) and that they were observing a point on local control panel A, mentioned in clause 9.41, at (6.0,6.5, 1.4) then the visual angle would be approximately 19 degrees below the horizontal. The panel would fall outside of the line of sight for comfortable eye rotation. This is derived from the following formula that relates the operator's seated eye height and the distance of a target on a control panel to the height of that target and the likely visual angle between the horizontal plane and that target:

$$\sqrt{seated\_height^2 + target\_distance^2}/sin\ 90 = target\_height/sin\ visual\_angle \tag{9.43}$$

Such formulae can be used to guide interface development. In particular, it can be used to ensure that operators can actually monitor and use the multiple displays of CSCW systems. Designers should not place routinely monitored information for $user\_1$ on the local control panel. The operator would be forced to assume an undesirable posture to observe the display. This area might be used to present information for other operators who can more easily view this display. Similarly, $user\_1$ cannot be expected to observe high-priority error messages on the local control panel. Operators frequently fail to detect warnings on the edge of their vision [Wic84]. The identification of such "high-priority" errors is an important stage during the development of safety-critical interfaces. $User\_1$'s observation problem with local control panel A might be resolved by ensuring that such critical warnings are also presented closer to their normal line of sight. Equation (9.43) can be used to validate $user\_1$'s line of sight between various positions in the control room and these additional sources of information. These positions must, in turn, be checked to ensure that they do not obscure critical information for $user\_2$, $user\_3$ etc. Once an optimal position has been identified, logic can be used to represent the new position for the display:

$$\begin{aligned} & resolve\_observation\_problem \Leftarrow \\ & \quad location(user\_1, 6.0, 2.2) \wedge \\ & \quad present(user\_1, Display\_1, local\_panel\_A) \wedge \\ & \quad part(user\_1, Display\_1, compressor\_failure) \wedge \\ & \quad present(user\_1, Display\_2, worktop\_panel) \wedge \\ & \quad part(user\_1, Display\_2, compressor\_failure). \end{aligned} \tag{9.44}$$

This states that a potential observation problem can be resolved if $user\_1$ is located at (6.0, 2.2) and they are allocated a display, $Display\_1$, which includes a warning that a compressor is failing and that display is presented on the local control panel and they are allocated a display, $Display\_2$, which also includes a warning that the compressor is failing and that display is presented on the $worktop\_panel$.

Workstation layout not only affects the presentation of control information, it also has a profound impact upon input requirements. For example, Grandjean uses Figure 9.11 to illustrate the working distance from the elbow to the hand of an operator at table-top height [Gra88]. This applies to the fifth percentile of the male population. The inner arc represents the extent of the grasp from a relaxed, seated position. This analysis can be used to inform dialogue design. For example, in control systems it is important that certain input sequences are dif-

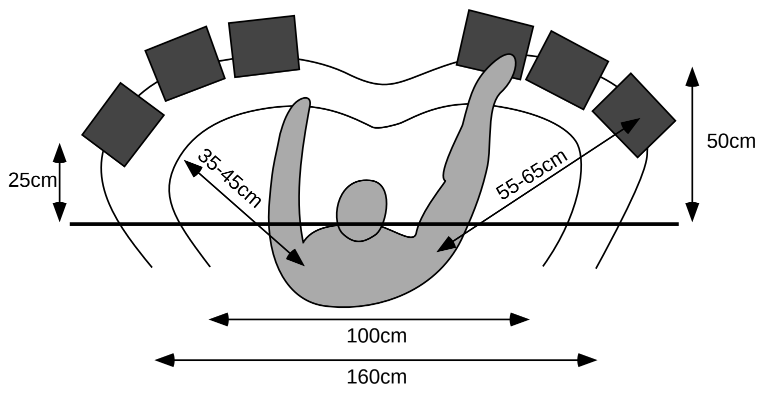


**Figure 9.11** The horizontal reach limit

ficult to issue. The valve isolation switches might, therefore, be placed beyond the 55–65cm arc. Operators can make occasional stretches of 70–80cm without difficulty:

$$
\begin{aligned}
&reach\_isolate\_valve\_A \Leftarrow \\
&\quad location(user\_1, 6.0, 2.2) \wedge \\
&\quad select(user\_1, close\_valve\_A\_switch) \wedge \\
&\quad component(close\_valve\_A\_switch, worktop\_panel) \wedge \\
&\quad location(close\_valve\_A\_switch, 6.0, 3.0, 0.9).
\end{aligned} \tag{9.45}
$$

This states that the user must reach to close off valve A if they are at (2.0, 2.1) and they provide input to isolate the valve by selecting a button on the worktop control panel at (6.0,3.0,0.9).

The correct positioning of control panel components must reflect details of the operators' tasks. It should be hard to issue input sequences that cannot easily be reversed. Conversely, the input requirements that are implicit within dialogue designs must also take into account the physical demands that devices place upon their users. Operators should not routinely be expected to sustain postures that impose significant biomechanical strain.

## 9.8 PROTOTYPING

Mathematical specifications provide users with little idea of what it would be like to interact with a graphical interface. Prototypes provide a far better impression of the "look and feel" of a potential implementation. The experimental analysis of partial implementations can be used to inform the refinement of detailed specifications towards full implementation. They can be shown to members of concurrent design teams. They can be shown to operators and are amenable to experimental analysis. Logic programming environments, such as that supported by PROLOG, provide a convenient bridge between formal specifications and functioning prototypes. This environment has a well understood semantics based on that of first-order logic. This secton, therefore, provides a brief introduction to the design and implementation of the

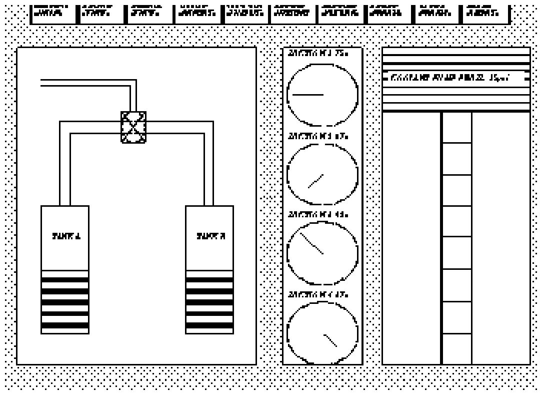


**Figure 9.12** A Prelog prototype

Prelog system. This application has been specifically developed to implement CSCW prototypes. Prelog combines a temporal logic interpreter and a screen management system to directly execute the clauses that have been presented in this chapter.

### 9.8.1 Presenting Graphical Structures

A number of research groups have developed executable versions of the temporal logic notation that has been used in this chapter. For instance, the Tokio interpreter has been implemented using the PROLOG programming environment [Aoy86]. Clauses that contain temporal operators are re-written and are asserted over an appropriate interval. In other words, the Tokio interpreter maintains time-variables that are similar to those introduced in the earlier sections of this chapter. Unfortunately, Tokio only provides limited input and output facilities. The Prelog prototyping tool avoids this limitation by linking Tokio and Presenter [Too91]. Prelog uses the Presenter screen management system to provide facilities for manipulating region structures and for setting, clearing and interrogating properties of regions. Low-level implementation details, such as raster graphics operations, are isolated within the presentation system. This is a significant benefit for the development of CSCW system. Designers are not forced to consider low-level details for multiple presentation devices. Figure 9.12 shows a display that was generated using Prelog. In order to produce such an image, Prelog constructs a $part$ hierarchy using clauses such as (9.21,9.22,9.23). The region properties, attributes and behaviors of each part, represented by clauses such as (9.33), are then recorded in a tree. This data structure is traversed. Information about each region is passed to Presenter. For designers, the net effect of linking Tokio and Presenter is to provide the impression of a graphical output channel.

### 9.8.2 Handling Device Input

Prelog must translate device primitives into input events. It is important that the complexity of handling input from many concurrent users should not frustrate the design of CSCW systems. Prelog reduces this complexity by isolating low-level device handling within Presenter. Current implementations support $on\_select$, $on\_move$ and $on\_size$ to represent initial selection, move and scaling events. The $on\_select\_up$, $on\_move\_up$ and $on\_size\_up$ events represent terminating selection, move and scaling. Users may type input directly into editable regions.

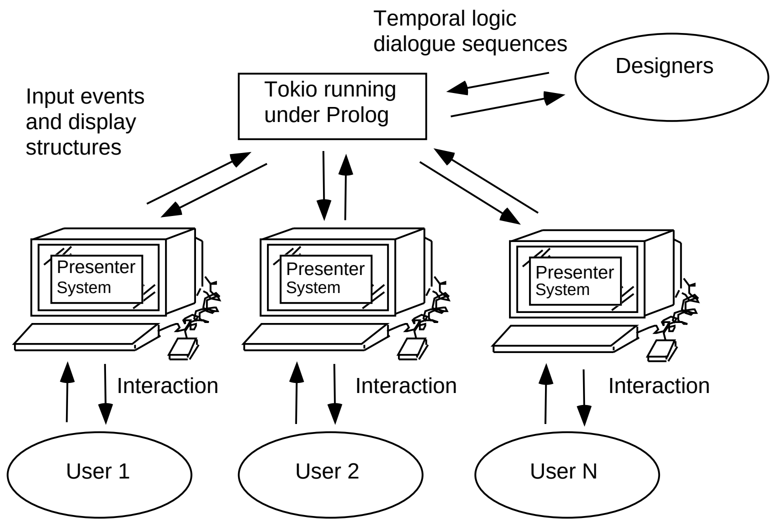


**Figure 9.13** The distributed Prelog architecture

Such keyboard input is represented by an $on\_CR$ event which is generated each time a carriage return is pressed. If necessary, designers can then specify that Prelog should read the text which has been entered into an editable region.

It can be extremely computationally expensive to support fine-grained updates of shared objects in CSCW systems. Presenter provides means of reducing this cost; it implements Took's notion of surface interaction [Too91]. Some graphical operations, such as textual and geometric manipulations, have no "deep" semantic meaning for an application. They can, therefore, be handled by Presenter without reference to the logic specification. For instance, designers can specify that the image of the pump presented to $user\_1$ changes under selection. Presenter will then automatically highlight the region whenever $user\_1$ selects it. The designer is only forced to explicitly request this image update if it is to be presented to other users. Prelog also provides efficiency features which ensure that certain input events from particular operators can be discarded. For instance, $on\_move$ and $on\_move\_up$ might be ignored by safety-critical CSCW systems in which users are prevented from altering the layout of their displays. All of these enhancements are optional and can be explicitly represented in the clauses of logic specifications.

Controlling event-based interfaces from within a logic programming environment raises many practical and theoretical problems. In particular, it is unclear how asynchronous, concurrent input from many different operators can be supported without sacrificing Prelog's notion of execution as proof. If Prelog is interrupted with new input events, how should this information be accommodated within an ongoing proof? For instance, if Prelog were forced to suspend a proof to handle a $move$ event on an inlet icon, it might have to ensure that prior proof steps did not depend on previous information about the position of that object. This would radically affect the nature of the programming environment provided by Prelog. A large number of input events might stretch the resources of any implementation to an unacceptable level. An obvious alternative is to make Prelog responsible for sampling input. The designer is free to specify when Prelog should poll Presenter. One drawback to this approach is that important events from one user can be stored until Prelog has finished handling less important input from other users.

Tokio was intended to run on single-user, single-processor implementations. In contrast, our implementation of Prelog uses UNIX sockets [Lef90] to support interaction between a number of users communicating over local and wide area networks. Figure 9.13 illustrates the Prelog architecture. In the current implementation of Prelog, $graphics\_write(To\_client, Message)$ is evaluated as true if a $Message$ string is successfully sent to the client process running on the user's workstation. $graphics\_read(From\_client, Message)$ is evaluated as true if $Message$ unifies with input sent from a client process. For example, $event\_close\_pump\_A$ (9.36) can be implemented by the following clause. The $graphics\_read$ and $graphics\_write$ formats are used here to aid the exposition. Designers can re-name these clauses to increase the tractability of an executable specification:

$$\begin{aligned}
&close\_pump\_A\_prototype \Leftarrow \\
&\quad graphics\_write(user\_1, display(condensate\_display)), \\
&\quad graphics\_write(user\_2, display(deluge\_display)), \\
&\quad graphics\_write(user\_1, part(condensate\_display, pump\_A\_error\_icon)), \\
&\quad graphics\_write(user\_2, part(deluge\_display, pump\_A\_error\_icon)), \\
&\quad \Diamond(graphics\_read(user\_1, input(pump\_A\_error\_icon, on\_select), \\
&\quad (not(effect(on\_select, pump\_A\_error\_icon, pump\_A\_off)\mathcal{U} \\
&\quad graphics\_read(user\_2, input(pump\_A\_error\_icon, on\_select)). \qquad (9.46)
\end{aligned}$$

This states that there is a dialogue to close $pump\_A$ if a message is written to $user\_1$'s client to ensure that they are presented with the $condensate\_display$ and a message is written to $user\_2$'s client to ensure that they are presented with the $deluge\_display$ and messages are sent to ensure that the $pump\_A\_error$ icon is part of the displays and eventually a select event for that icon is read from the $user\_1$. This input is ineffective until a selection is also read from $user\_2$.

Prelog also supports the implementation of CSCW systems which provide multiple windows on each workstation. A stub process is created for each window, graphical clauses are easily parameterized by their intended destination; $user\_1\_window\_1$.

### 9.8.3 Environmental Animation

Prelog offers significant advantages over traditional prototyping tools. Previous systems help developers to quickly mock-up CSCW displays and animate dialogue sequences. There is a danger, however, that such tools may produce dialogues which cannot easily be integrated with their eventual working environment. Warnings may be obscured by other operators or pieces of equipment. On-line help may be abandoned if users cannot easily read particular displays. In contrast, the Prelog tool exploits $location$ clauses such as those in (9.45) to build up three-dimensional models of control rooms and offices. The same system can, therefore, be used to prototype dialogues as well as view the potential layout of working environments. These models can be shown to operators and to the members of concurrent design teams that are working on control room planning and display development. The term "environmental

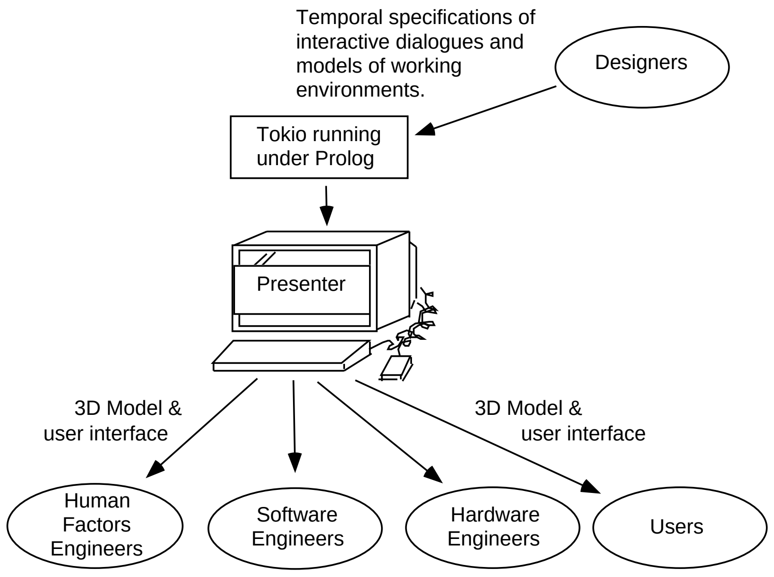

Temporal specifications of interactive dialogues and models of working environments.
Designers
Tokio running under Prolog
Presenter
3D Model & user interface
3D Model & user interface
Human Factors Engineers
Software Engineers
Hardware Engineers
Users

**Figure 9.14** The application of Prelog for environmental animation

animation" has been used to refer to our integration of prototyping techniques and three-dimensional models. Figure 9.14 illustrates this aspect of the Prelog architecture. Further work intends to explore the more general use of formal notations to reason about the physical characteristics of working environments. For instance, logic can also be used to characterize acoustic properties. Layout information might then be recruited to represent appropriate sound levels within particular areas of a control room. Prelog might provide rudimentary simulations for these presentation techniques:

$$timbre(pump\_A\_error\_alarm, bell). \tag{9.47}$$

$$amplitude(pump\_A\_error\_alarm, 60dBA). \tag{9.48}$$

$$pitch(pump\_A\_error\_alarm, 260Hz). \tag{9.49}$$

This states that the $pump\_A$ warning has the timbre of a bell, the amplitude of the warning is 60dB and its pitch is 260Hz.

In many human–machine interfaces, changes in the characteristics of acoustic signals are used to indicate changes in the underlying state of an application. For instance, a continuous tone might change into a bell in order to indicate a failure in the deluge system. Temporal logic offers one means of explicitly representing these dynamic properties [Joh91, Joh90].

## 9.9 CONCLUSION

This chapter has shown how mathematical specification techniques can support the design of CSCW systems. In particular I have argued that temporal logic can be used to represent

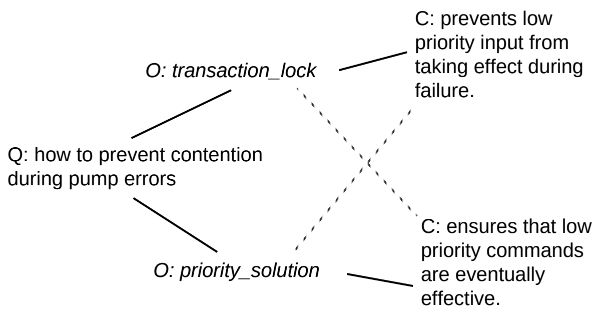

C: prevents low priority input from taking effect during failure.
O: transaction_lock
Q: how to prevent contention during pump errors
C: ensures that low priority commands are eventually effective.
O: priority_solution

**Figure 9.15** Literate specification for $transaction\_lock$

critical requirements for sychronization and locking. This approach is justified because temporal properties have a profound impact upon the nature of interaction in multi-user systems. It has also been argued that graphical information and input events must be explicitly represented within abstract models of CSCW applications. I have also shown how the application of formal methods can be extended to represent and reason about the physical dimensions of working environments. This is often neglected within the development of CSCW systems and represents an important extension to the application of formal methods. For CSCW systems, the layout of an office, factory or control room will have a critical impact on the operation and use of a human–computer interface. Finally, I have argued that prototyping tools must be provided if non-formalists are to assess the products of mathematical specification techniques. The Prelog system has been developed to address this concern. It can be used to directly derive partial implementations from temporal logic clauses, such as those introduced in this chapter. It can also be used to generate environmental animations, or 3-D models, of potential workstation layouts. This enables designers to view potential displays within their intended context of use.

Many questions remain to be addressed before formal methods can be widely applied to support the development of multi-user systems. In particular, there are problems in scaling up the approach to deal with very large-scale systems. Such applications raise a different set of CSCW problems. Not only do they raise issues about synchronizing multi-user access to computer resources, these design challenges also force designers to consider the synchronization of multiple development teams. This is difficult because many members of these teams will have no understanding of formal methods. In order to address this issue, we are developing literate specification techniques [Joh96a, Joh95b]. This approach provides clients and users with access to both formal and semi-formal documentation. In particular, design rationale is used to record the reasons *why* particular clauses were used during the development of a CSCW specification. For example, Figure 9.15 presents the arguments for and against locking out the user in the manner described by $transaction\_lock$ (9.16) and $priority\_solution$ (9.14). The problem of reducing contention during pump failures can either be satisfied by transaction locking or by the use of input priorities. These are labelled as alternative options, $O$. These options are linked to criteria, $C$, which represent the reasons for and against a particular approach. In the case of transaction locking this is supported by the criteria that it prevents low priority input from taking effect during the failure. Positive or supporting criteria are indicated by solid lines. In contrast, it is not supported by the argument that low-priority commands will eventually be effective. This is because they are literally locked-out of the system. The dotted lines indicate negative or weakening criteria. The intention here is that the QOC argumentation structures should enable designers to question the approaches that are embodied within formal specifications. It should be possible for non-formalists to ask *why* a system is designed the way it is.

The literate specification approach, described above, addresses a fundamental paradox in the formal design of CSCW systems. In order to obtain precise notations for reasoning about the complexity of multi-user communication, we may lose the ability to communicate within and between multiple design teams. Not everyone can be expected to learn and understand temporal logics. This re-iterates a key point for future work in this area. We will not be able to develop interfaces that support groupwork unless we provide techniques that can be used by groups of designers. This represents the greatest challenge to the continued application of formal methods for CSCW systems.

## REFERENCES

[App86] Rose, C. *Inside The Apple Macintosh*, Vol. I. Addison Wesley, Wokingham, UK, 1986.

[Aoy86] Aoyagi, T., Fujita, M. and Moto-Oka, T., Temporal logic programming language -Tokio-programming in Tokio. In Wada, E. (Ed.), *Proceedings of the 4th Annual Conference - Logic Programming '85*, LNCS 221, pages 128–137. Springer-Verlag, Berlin, Germany, 1986.

[Bas90] Bastide, R. and Palanque, P., Petri net objects for the design, validation and prototyping of user-driven interfaces. In Diaper, D., Gilmore, D., Cockton, G. and Shackel, B. (Eds.), *Human–Computer Interaction — INTERACT'90*, pages 625–631. Elsevier Science Publications, North Holland, Netherlands, 1990.

[Bur89] Burton, C.T., Cook, S.J., Gikas, S., Rowson, J.R. and Sommerville, S.T., Specifying the Apple Macintosh Toolbox Event Manager. *Formal Aspects of Computing*, 1:147–171, 1989.

[Cra93] Craigen, D., Gerhart, S. and Ralston, T., 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, USA, 1993.

[Cul90] Cullen, *Proceedings of the Public Enquiry into the Piper Alpha Disaster*. The Department of Energy, London, UK, 1990.

[Dix97] Dix, A., Rodden, T. and Sommerville, I., Modelling versions in collaborative work. *IEE Proceedings in Software Engineering*, 14(4):195–205, 1997.

[Ell89] Ellis, C.A. and Gibbs, S.J., Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[Ell91] Ellis, C.A., Gibbs, S.J. and Rein, G.L., Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):35–58, January 1991.

[Gal80] Galer, I.A.R. and Yap, B.L., Ergonomics in intensive care: Applying human factors to the design and evaluation of a patient monitoring system. *Ergonomics*, 23(8):763–779, 1980.

[Gra88] Grandjean, E., *Fitting the Man to the Task: Occupational Ergonomics*. Taylor & Francis, London, UK, 1988.

[Gra95] Gray, P.D. and Johnson, C.W., Requirements for interface design notations. In Palanque, P. and Bastide, R. (Eds.), *Design, Specification and Verification of Interactive Systems '95*, pages 113–133. Springer Verlag, Berlin, Germany, 1995.

[Gre87] Greif, I. and Sarin, S., Data sharing in group work. *Communications of the ACM*, 5(2):197–211, 1987.

[Har95] Harrison, M.D., The role of verification. In Palanque, P. and Bastide, R. (Eds.), *Design, Specification and Verification of Interactive Systems '95*, pages 342–344. Springer Verlag, Berlin, Germany, 1995.

[Hix93] Hix, D. and Hartson, H.R., *Developing User Interfaces*. John Wiley & Sons, London, 1993.

[Hod77] Hodges, W., *Logic*. Penguin Books, London, 1977.

[Joh90] Johnson, C.W. and Harrison, M.D., Using temporal logic to support the specification and prototyping of interactive control systems. *International Journal of Man–Machine Studies*, 36:357–385, 1992.

[Joh91] Johnson, C.W., Applying temporal logic to support the specification and prototyping of concurrent multi-user interfaces. In Diaper, D. and Hammond, N. (Eds.), *People And Computers VI: Usability Now*, pages 145–156. Cambridge University Press, Cambridge, UK, 1991.

[Joh92] Johnson, C.W., Specifying and prototyping dynamic human-computer interfaces for stochastic applications. In Alty, J.L., Diaper, D. and Guest, S. (Eds.), *People And Computers VIII*, pages 233–248. Cambridge University Press, Cambridge, UK, 1993.

[Joh94a] Johnson, C.W., McCarthy, J.C. and Wright, P.C., Using a formal language to support natural language in accident reports. *Ergonomics*, 38(6):1265–1283, 1995.

[Joh94b] Johnson, C.W., Representing and reasoning about the impact of environmental layout upon human computer interaction. *Ergonomics*, 39(3):512–530, 1996.

[Joh95a] Johnson, C.W., Using Z to support the design of interactive, safety-critical systems. *IEE Software Engineering Journal*, 10(2):49–60, 1995.

[Joh95b] Johnson, C.W., Literate specification. *Software Engineering Journal*, 11(4):224–237, 1996.

[Joh96a] Johnson, C.W., Documenting the design of safety-critical user interfaces. *Interacting With Computers*, 8(3):221–239, 1996.

[Joh96b] Johnson, C.W. and Gray, P.D., Error driven design. In Harrison, M.D. and Vanderdonk, J. (Eds.), *Design, Specification and Verification of Interactive Systems'96,* Springer Verlag, Berlin, Germany, 1996.

[Joh97] Johnson, C.W., The impact of time and place on the operation of mobile computing devices. In *People and Computers XII*, pages 175–190. Springer Verlag, Berlin, Germany, 1997.

[Kan88] Kantowitz, B.H. and Casper, P.A., Human workload in aviation. In Wiener, E.L. and Nagel, D.C. (Eds.), *Human Factors In Aviation*, pages 157–187. Academic Press, London, UK, 1988.

[Kra91] Kramer, B., Introducing the GRASPIN specification language SEGRAS. *Journal of Systems and Software*, 15(1):17–31, 1991.

[Kuh89] Kuhmann, W., Stress inducing properties of system response times. *Ergonomics*, 32(3):271 – 280, 1989.

[Lef90] Leffler, S.J., McKusick, M.K., Karels, M.J. and Quarterman, J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, Reading, USA, 1990.

[Mal89] Malone, T.B., MPTS methodology in the Navy: Enhanced HARDMAN. In Pettigrew, D.L. (Ed.), *Proceedings of the 33rd Annual Meeting of the Human Factors Society*, pages 1044–1048. Human Factors Society, Santa Monica, USA, 1989.

[Man81] Manna, Z. and Pnueli, A., Verification of concurrent programs: The temporal framework. In Boyer, R.S. and Strother Moore, J. (Eds.), *The Correctness Problem In Computer Science*, pages 215–273. Academic Press, London, UK, 1981.

[McC91] McCarthy, J.C., Miles, V. and Monk, A.F., An experimental study of common ground in text-based communication. *Proceedings of the CHI'91 Conference on Human Factors in Computing Systems*, pages 209–215. ACM, New York, USA, 1991.

[Pal95] Palanque, P. and Bastide, R., Formal specification and verification of CSCW using interactive cooperative object formalism. In *People and Computers X*, pages 197–212. Springer Verlag, Berlin, Germany, 1995.

[Pen90] Pendergast, M.O. and Vogel, D., Design and implementation of a P.C. based multi-user text editor. In Gibbs, S. and Verrijn-Stuart, A.A. (Eds.), *Multi-User Interfaces And Applications*, pages 195–206. Elsevier Science Publications, North Holland, Netherlands, 1990.

[Pri67] Prior, A., *Past, Present, Future*. Oxford University Press, Oxford, UK, 1967.

[Sze88] Szekely, P. and Myers, B., A user interface toolkit based on graphical objects and constraints. *ACM SIGPLAN Notices*, 23(11):36–45, 1988.

[Too91] Took, R., Integrating inheritance and composition in an objective presentation model for multiple media. In Post, F.H. and Barth, W. (Eds.), *EUROGRAPHICS '91*, pages 291–303. Elsevier Science Publications, North Holland, Netherlands, 1991.

[War89] Wardell, R.W., An ergonomics perspective on safety in the oilfield. In Pettigrew, D.L. (Ed.), *Proceedings of the 33rd Annual Meeting of the Human Factors Society*, pages 999–1003. Human Factors Society, Santa Monica, USA, 1989.

[Wic84] Wicken, C.D., *Engineering Psychology And Human Performance*. C.E. Merrill Publishing Company, London, UK, 1984.

[Win87] Winograd, T. and Flores, F., *Understanding Computers And Cognition*. Addison-Wesley, Reading, USA, 1987.