Using Temporal Logic To Support The Specification And Prototyping Of Interactive Control Systems

C.W. Johnson and M.D. Harrison,
H.C.I. Group, Department of Computer Science,
University of York, United Kingdom, YO1 5DD.
Phone: (0904) 432765. Fax: 432767.
Email: johnson@minster.york.ac.uk

Abstract

Accidents at Flixborough, Seveso, Bhopal, Three Mile Island, Windscale and Chernobyl have led to increasing concern over the safety and reliability of control systems. Human factors specialists have responded to this concern and have proposed a number of techniques which support the operator of such applications. Unfortunately, this work has not been accompanied by the provision of adequate tools which might enable a designer to carry it beyond the 'laboratory bench' and onto the 'shop floor'. The following paper exploits formal, mathematically, based specification techniques to provide such a tool. Previous weaknesses of abstract specifications are identified and resolved. In particular, they have failed to capture the temporal properties which human factors specialists identify as crucial to the success or failure of interactive control systems. They also provide the non-formalist with an extremely poor impression of what it would be like to interact with potential implementations. Temporal logic avoids these deficiencies. It can make explicit the sequential information which may be implicit within a design. Executable subsets of this formalism support prototyping and this provides a means of assessing the qualitative 'look and feel' of potential implementations. A variety of presentation strategies, including structural decomposition and dialogue cycles, have been specified and incorporated directly into prototypes using temporal logic. Prelog, a tool for the Presentation and REndering of LOGic specifications, has been developed and its implementation is described.

Keywords: *Interface Design, Temporal Logic, Prototyping, Specification.*

1 Introduction

There is a growing concern over the safe operation of control systems. This has arisen in the aftermath of accidents in both the chemical and nuclear production industries at Flixborough, Seveso, Bhopal, Three Mile Island, Windscale and Chernobyl. Many of these accidents have been caused or exacerbated through human error (Stephens 1980, Green 1988). The risks and consequences of such error can be reduced by producing interfaces which support the operator of safety-critical applications. Human factors specialists and, in particular, cognitive and perceptual psychologists have suggested a number of presentation techniques which might be adopted to satisfy this requirement. Unfortunately, many of these advances have not been carried through into working systems. Part of this failure is due to the lack of adequate tool support for the designer who must realise these ideas within an implementation. The following pages argue that formal, mathematically based, specification techniques fulfill this omission. In particular, it is argued that temporal logic is appropriate for this task because it can capture critical timing properties of interactive control systems which can then be visualised using executable subsets of the formalism It provides a means of extending the utility of formal methods beyond systems and software engineering to support human factors engineering.

1.1 Human Factors And Design

The last twenty years have seen significant advances towards an understanding of the cognitive and perceptual demands which control systems place upon the finite resources of their operators. For example, Rasmussen (1983) has developed a skill, rule and knowledge based classification of the informational requirements for control. Reason (1987) and Singleton (1973) have proposed explanations of human error. Woods (1988) has identified the causes and consequences of task complexity. Moray and Rotenberg (1989) have investigated the perceptual demands of monitoring large amounts of dynamic information. Attempts have also been made to apply this research within complex interfaces. Notable examples include Beltracchi's (1989) reactor displays and the flight predictor produced by the United States Air Force's Flight Dynamics Laboratory (Hawkins, Reisning, Lizza and Beachy 1983). Unfortunately, many of these systems have failed to make the transition between 'laboratory bench' and 'shop-floor' (Alty, Elzer, Holst, Johannsen and Savory, 1986).

There are a number of potential explanations for the breakdown in technology transfer between human factors research and commercial interface design. Managerial considerations and the demands of process engineering are often perceived to outweigh the benefits of cognitive and perceptual psychology (Clegg and Wall 1987). Gaines (1988) argues that human factors models have failed to keep pace with commercial developments and that this has limited their application. It can also be argued that part of this breakdown is due to the conflicts inherent between human factors and system development. For example, systems design increasingly attempts to delay commitment to hardware platforms in order to benefit from competitive tendering. In contrast, human factors research encourages an early consideration of ergonomics and device characteristics which have a profound impact upon the usability of an interface. Similarly, system design has sought to abstract away from issues such as the timing of information presentation and structure of control-room displays. Human factors research has argued that these issues can determine the safety of an application. The following sections suggest a means of resolving this conflict of interests. Abstract specifications can be extended from system design to capture human factors requirements. Tools can be provided to support the rapid transformation of specifications into prototypes which validate the innovations of cognitive and perceptual psychology.

1.2 Specification And Prototyping

Many engineering disciplines use models to support the design of complex artifacts. These models offer a number of advantages for the design of interfaces to safety-critical systems. They provide a precise

and concise means of specifying human factors requirements. Characterising interaction in terms of mathematical or logical relationships strips away the complexity of implementation details, such as device handling, which obscure important properties of an interface early in the development process. It is important, however, that the application of formal design techniques should not obscure or abstract away from the temporal properties of control systems. These can frequently determine the success or failure of an interface, delays in system responses lead to frustration and error (Kuhmann 1989). The following pages argue that the representation of temporal information can be extended from the domain of Artificial Intelligence (AI) research and concurrency theory to support the development of interactive control systems. For example, temporal logic extends the static declarative power of first order logic with a means of reasoning about change over time. It provides a tractable medium in which to represent and reason about the dynamics of control. It can be incorporated into high level models of interaction and this provides a structure for the specification of interfaces to complex applications.

Many techniques proposed by cognitive and perceptual psychology rely upon presenting users with particular display formats or dialogue sequences. These techniques are not readily visualised from formal specification which are often terse and mathematical. Prototypes provide a much better impression of the 'look and feel' of control systems and these can be shown to users as a means of validating a design (Alexander, 1987). In practice, specification and prototyping are usually treated as alternatives. The following paper describes how these complementary approaches can be integrated. This integration involves the introduction of device and presentation details, such as mouse handling and screen updates, into high level models of control systems. These details do not normally form any part of mathematical specifications and can threaten the tractability of a design. Device abstractions and structured graphics systems provide a means of avoiding this undue complexity and have been incorporated into a prototyping tool, Prelog. Specifications can be directly executed because Prelog also includes a temporal logic interpreter.

1.3 Dialogue Cycles and Structural Decomposition

The dichotomy of interest between system design and human factors is not as great as it might appear. For instance, both disciplines are faced with the problems of complexity and dynamism. Human factors specialists are concerned to filter complex data in order to present relevant and timely information to system operators. Process engineers are concerned to represent and reason about this complexity in order to verify safety and reliability properties of process components. For example, the speed of chemical reactions and side-effects in pigment production complicates the systems engineering of computerised control applications. Changes in the production process can cause several variables to stray at once and action to correct one variable normally affects others. This complexity also influences the operation of control systems. More than twelve months training is necessary before operators can learn to effectively monitor and control such applications (Buchanan and Bessant, 1985). This paper argues that system design techniques can be extended to support the application of presentation strategies developed in human factors research. One way of testing this hypothesis is to apply specification techniques derived from engineering disciplines to investigate the utility of techniques proposed by human factors specialists.

Structural decomposition and *dialogue cycles* have been advocated as means of reducing the complexity of control. For example, Bainbridge (1988) argues that operators optimise scarce perceptual and cognitive resources by exploiting pattern recognition and cued memory techniques. Dialogue cycles provide a practical application of these observations by emphasising recurrent patterns during interaction. Displays, called gates, are used to mark the beginning and end of critical sections during interaction such as process errors (Harrison, Roast and Wright, 1989). Operators can maximise their allocation of finite monitoring resources by recognising cycles in behaviour rather than the detailed changes of particular variables. Qualitative process theory argues that structural information is also vital for the interpretation of system behaviour (Forbus 1987). Rasmussen and Lind (1981) maintain that structural decomposition supports control and fault diagnosis by presenting complex plant in terms of simpler component parts.

Unfortunately, whilst such techniques ease control tasks for system operators they provide little support for the interface designer. They are faced with the task of selecting critical cycles from the mass of detail which may be monitored by the control system. They must also determine and maintain structural decompositions which are liable to change during component failure or through flexible manufacturing techniques. The following sections use the examples of structural decomposition and dialogue cycles to show how temporal logic supports the performance of these design tasks.

1.4 Outline Of The Paper

The remainder of the paper is structured as follows. Section two describes how abstract models of interaction provide a structure which supports the specification of interface requirements for complex systems. The limitations of these models are described and techniques for capturing timing information are presented . Temporal logic is then advocated as a particularly tractable means of expressing dynamic requirements within a formal model of interaction. Section three demonstrates how high level models, constructed using temporal logic, can be applied to investigate presentation techniques proposed by human factors specialists. The potential benefits of *dialogue cycles* and *structural decomposition* are identified using these techniques. Further limitations of formal specifications are then described and section four introduces the justifications for prototyping. The problems of incorporating implementation and interaction details into high level specifications are considered and solutions are presented. This section also describes the implementation of Prelog. This tool supports the Presentation and REndering of LOGic specifications of interactive systems. It successfully integrates a structured graphics system and a temporal logic interpreter to support specification and prototyping of interactive control systems.

2 Specification

Abstract, mathematically based specifications have long been exploited in the construction and engineering industries for which complexity and dynamism are commonplace problems. The perceived benefits of such techniques include a tractability and precision not found in vernacular or informal descriptions of potential designs. Formal specifications support reasoning which can be employed to analyse important structural properties of an artifact before it is implemented. The application of these techniques has been encouraged by procurement guidelines such as the United Kingdom Ministry of Defence Interim Defence Standard 00-55. The application of formal specifications is gradually being extended to include the design of human-computer interfaces (Harrison and Dix 1987, Johnson 1990, Suffrin and He 1990). The following section describes how a high level model can be applied to support the development of complex interactive control systems. In order to specify such dynamic applications it must be possible to express their temporal properties. Approaches capable of incorporating such information into abstract specifications are described. Their strengths and weaknesses are explored. Finally, temporal logic is advocated as an expressive medium in which to construct interfaces for safety-critical control systems.

2.1 A High Level Model Of Control

Formal models of interaction offer the interface designer a descriptive framework which can be used to structure the design of interfaces to complex systems. By characterising interaction in terms of a number of high level abstractions they clear away the clutter of hardware requirements and presentation details. Interaction may be described using four sets: P which contains all possible input sequences; C which contains all possible commands; S which contains all possible states of the system and D all possible displays which a system might present. For example, a control system might interpret input, such as button presses and keystrokes, as commands to stop, *stoprun*, or start a system, *startrun*. States of the system might include *on*, *off* and *error*. The state of the system might be displayed using graphical

or audio output: *ondisp*; *offdisp* and *errordisp*. For example, *ondisp* might be presented whenever the system is in the *on* state.

Relationships between the components of the model can be specified using first order logic. This offers a number of advantages for the interface designer. First order logic is both abstract and declarative. A designer can describe the structure of a problem without being forced to consider low-level implementation details. It also provides a common language for the specification and prototyping of interactive systems (Johnson 1990a, Johnson and Harrison 1991). There is a close correspondence between specifications written in first order logic and programs which satisfy those specifications implemented in Prolog (Clark, 1981). Hence the designer has a means of constructing and evaluating designs expressed in terms of high level abstractions rather than low level device input and unstructured bitmaps (Johnson and Harrison, 1990).

First order logic can be used to define an interpretation relation between all sequences of physical input, the state in which they are interpreted and unique elements of C:

$$\forall p \in P, \forall s \in S, \exists! c \in C : \\ interpret(p, s, c)$$

For all input, p, there exists a unique command, c, such that the interpretation of p in s is c.

In the above, *interpret* provides an interpretation, c, for input p in state s.

It is possible to trace interaction as state transitions triggered by input into the system. The effect of a command may be expressed in terms of a pre-condition (i.e., the state in which the command is received by the system) and a post-condition (i.e., the state of the system after the command is acted upon). For instance, an *effect* relationship can be defined between each element of C and pairs of states:

$$\forall c \in C, \forall s \in S, \exists! s' \in S : \\ effect(c, s, s')$$

The effect of all commands, c, is to transform the state of the system from s to a unique state s'.

Intuitively, s might be thought of as the context, and s' as the result of the command. A designer may consider a number of possible effects for each command in a particular state. We require a single effect, s', in order to retain the determinism necessary to support the implementation of prototypes.

The appearance of a system can be incorporated into this generic model of interaction by defining a relation between all states and elements of D:

$$\forall s \in S, \exists d \in D : \\ view(s, d)$$

All states, s, are viewed through a display, d.

$View$ describes a relationship between states and the displays which could be used to present them. Figure 1 illustrates the relationships between P, C, S and D.

We can build more complex relationships from those described above. For instance, we can describe the reception, interpretation and effect of input as follows:

$$\forall p \in P, \forall s, s' \in S : inputeffect(p, s, s') \Leftrightarrow$$
$$\exists! c \in C \ (state(s) \wedge input(p) \wedge$$
$$interpret(p, s, c) \wedge effect(c, s, s'))$$

The effect of all input, p, on all states, s, is s' iff the state of the system is s and there exists a unique command c which is the interpretation of p in s and the effect of c on s is s'.

The following section uses this abstract, generic model of interaction to provide a structure for the specification of interfaces to control applications.

2.2 An Application

The abstract model, presented above, can be instantiated to structure an interface design for a continuous casting plant. This application exhibits many of the problems which complicate the design of interactive control system, it is both complex and dynamic. System designers must regulate the operation of the plant by producing accurate assessments of process state based on over one hundred input values. Casting plants have also posed a significant challenge for human factors specialists (De Keyser 1987, Hoc 1987). The control of this production process, illustrated by figure 2, provides the example for the remainder of the paper. Steel from a melting shop is cast and then poured into moulds. The steel is passed through a water cooling mechanism and, in so doing, solidifies into a billet. This is cut to length at the end of the run. During this process the most feared accident is a break-out during which the billet tears causing the steel to flow inside the caster. There is also a danger that the flow will clog leading to a splash-off either at the tundish or onto the floor. Casting must halt if either a break-out or a splash-off are detected. Thermocouple monitors provide error warnings by alerting the system of increases in temperature. These reports can be cancelled by confirmation that the error has been dealt with.

2.3 Applying The Model

Interactive systems interpret input as commands. In the casting example a user might issue a command to start a run by typing the input sequence *start*. This can be expressed using the abstract model of interaction presented in previous sections:

$$interpret('start', off, startrun)$$

The input sequence *start* is interpreted in the *off* state of the system as a command to start a run.

Clauses, like that given above, are both language and implementation independent. The first order logic description provides an initial specification which must be refined to capture the details of input sequences, commands and states prior to implementation. Such abstract clauses provide a high-level architecture for any subsequent modelling of an interface. Once these details have been instantiated the abstract specification can be realised in one of a number of implementation or prototyping languages. The designer is not forced to specify the particular devices which an operator might employ to issue the input *start*. It might be generated by mouse selection, by pressing a touch screen or by input from a keyboard. The choice of input medium is liable to have a considerable impact upon the usability of an

interface. Delaying commitment to a hardware implementation leaves the designer free to explore "the design space of input devices" (Card, Mackinlay and Robertson 1990).

The state of the system is included as an additional parameter in the previous clause because the interpretation of any input may change depending on the state of the system. For instance, it may be invalid to attempt to start a run which is already on:

$$interpret('start', on, invalid)$$

The input sequence *start* is interpreted in the *on* state of the system as an invalid command.

Badly formed or incomplete input sequences, for instance *star* instead of *start*, may also be interpreted as invalid. The use of abstract specifications avoids the necessity of specifying the details of states or the mechanisms employed to implement commands early in the development cycle. The main advantage of this is that graphical and behavioural details can be added gradually as design progresses.

The effect of commands can be specified in a similar manner. For example, a designer might specify that invalid commands do not change the state of a working system:

$$effect(invalid, on, on)$$

An invalid command in the *on* state is ineffective.

This illustrates how abstractions can be used to hide presentation and implementation details. This bare-bones requirement provides a means of constructing a design without necessarily specifying all the error messages which a system would employ to warn a user of ineffective commands. These details may be captured during subsequent refinements to the high level model. This abstraction process also hides the complex state transitions which might occur when such warnings are provided. Effective commands can be described in similar fashion. For instance, a command to start a run may have the effect of changing a system from the *off* state into the *on* state:

$$effect(startrun, off, on)$$

The effect of a command to start a run in the off state is to change the system to the *on* state.

This illustrates how generic models of interaction can be instantiated to describe the functional requirements which must be fulfilled by systems engineers. Clauses such as *effect* make explicit the application functionality which may be an implicit assumption within the design of an interface.

It is vital that there be some means of capturing the appearance of an interface if human factors requirements are to be integrated into specifications of interactive control systems. For instance, a particular display might be used to present certain information about the state of a system:

$$view(error, errordisp)$$

This states that *errordisp* is a view of the *error* state.

Such descriptions avoid commitment to particular output devices or to the exact details of the state of a system. They avoid the introduction of device primitives, such as bitmaps, which can increase the complexity of design. They can be directly incorporated into dialogue specifications.

The components of the generic model can be combined using first order logical operators to describe more complex requirements. For example, that the input *start* has the effect of turning the off state *on*:

$$turnon \Leftarrow state(off) \wedge input('start') \wedge$$
$$interpret('start', off, startrun) \wedge effect(startrun, off, on)$$

A system is started if it is off and the interpretation of input *start* in the off state is the command *startrun* and this has the effect of turning the system *on*.

Given the previous definition of *inputeffect* this may be expressed more succinctly as follows:

$$simpturnon \Leftarrow inputeffect('start', off, on)$$

A system is started if the effect of input *start* is to turn the off state *on*.

Unfortunately, whilst such clauses simplify the complexity of designing interactive control systems, they completely fail to capture the temporal information which human factors specialists identify as vital to the usability of an interface.

2.4 Introducing Time Into The Model

Previous sections have shown that first order logic can abstract away from the device oriented complexity of interface design. Unfortunately, there are a number of limitations which restrict the application of these techniques to support the development of dynamic control systems. Lack of sequencing poses particular problems for first order logic specifications of interactive dialogues. For example, a designer might specify that casting is stopped safely if the change in state is displayed to the operator:

$$safeturnoff \Leftarrow inputeffect('stop', on, off) \wedge$$
$$view(off, offdisp) \wedge display(offdisp)$$

Casting is stopped safely if the input *stop* has the effect of turning the system off and this is presented to the operator by $offdisp$.

This clause would be satisfied if $offdisp$ were displayed before the system was in the off state. The logic specification does not determine the order in which $inputeffect('stop', on, off)$ or *display(offdisp)* are to be evaluated. Such an ordering is not only crucial to the description of dialogue sequences but also hinders the derivation of prototypes from abstract specifications. Logic programming languages must enforce an order of evaluation which is not explicit within declarative clauses. For example, the left-right strategy of Prolog would display *errordisp* before the input was received, the input would take *effect* before *offdisp* were presented. This introduces a distinction between the logic of a specification and the control which implementations require in order to determine an order of evaluation (Kowalski, 1979). An implementation of the previous clause would only exhibit the 'desired' behaviour because control considerations have influenced the ordering of the logical specification. Changing the order of $inputeffect('stop', on, off)$ and *display(offdisp)* would have no effect on the specification of a dialogue but would radically affect the behaviour of a prototype. This introduces significant complexity into the refinement necessary to implement high level logic specifications. The ordering of clauses within such a specification may have

to be substantially revised to support the control strategy of an implementation. This problem must be overcome if logic programming is to provide a convenient bridge between prototyping and specification, between human factors and systems engineering.

2.4.1 Time Points

The lack of sequencing in first order logic can be overcome by the introduction of temporal information into logic specifications of interactive systems. Dialogue sequences and the order of evaluation may be described with respect to an interval of time. Such an ordering can be introduced in a number of ways. For example, Langefors and Sundgren (1975) describe 'elementary records' which introduce time points into each piece of world knowledge:

$$pointturnoff(12.00.05) \Leftarrow inputeffect('stop', on, off, 12.00.00) \wedge$$
$$view(off, offdisp, 12.00.05) \wedge display(offdisp, 12.00.05)$$

Casting is stopped at five seconds past midday if the input *stop* has the effect of turning the system off at midday and $offdisp$ is presented at five seconds past midday.

Unfortunately, the expression of non-trivial timing properties involves considerable complexity and requires the introduction of many temporal parameters (Bolour, Anderson, De Keyser and Wong 1982). In particular, time points are ill-suited to the specification of persistent properties of interaction. For example, a designer might specify that *ondisp* should continue to be presented until the operator is informed that casting has been halted:

$$persiturnoff(12.00.03) \Leftarrow inputeffect('stop', on, off, 12.00.00) \wedge$$
$$display(ondisp, 12.00.01) \wedge display(ondisp, 12.00.02) \wedge$$
$$display(ondisp, 12.00.03) \wedge display(ondisp, 12.00.04) \wedge$$
$$view(off, offdisp, 12.00.05) \wedge display(offdisp, 12.00.05)$$

Casting is stopped at five seconds past midday if the input *stop* has the effect of turning the system off at midday and *ondisp* continues to be presented until $offdisp$ is presented to the operator at five seconds past midday.

Specifications constructed using fixed time points impose unrealistic demands upon system operators. In order to fulfill the previous requirement a user must input *stop* at midday exactly! This is likely to be an extremely optimistic assumption given finite cognitive resources and divided attention in noisy environments.

2.4.2 Time Variables

The temporal determinism of fixed time points can be avoided by substituting time variables for explicit parameters. For example, the previous definition of $pointturnoff$ might be expressed as follows:

$$varturnoff(T) \Leftarrow inputeffect('stop', on, off, T1) \wedge$$
$$view(off, offdisp, T) \wedge display(offdisp, T) \wedge after(T, T1)$$

Casting is stopped at some time T if the input *stop* has the effect of turning the system off at time $T1$ and $offdisp$ is presented to the operator at T and T occurs after $T1$.

Clausal ordering is made explicit by the predicate *after*. The use of temporal variables can still introduce considerable complexity into a design (Gabbay 1987). In particular, designers must maintain a clear semantics for predicates, such as $after$, which define an ordering over time variables. These semantics can radically affect the properties of any specification.

2.4.3 Temporal Logic

In contrast to time points and variables, temporal logic provides a concise and precise means of describing complex properties of dynamic systems. It possesses a well founded semantics based on that of modal logic which has already been exploited to support the design of process control applications (Sakuragawa 1987). It supports reasoning about concurrency (Manna and Pnueli 1981). It provides the designer with proof techniques through Kripke semantics. Temporal logic extends first order logic to include the following operators: $\Diamond$ (read as 'eventually'); $\bigcirc$ (read as 'next'); $\Box$ (read as 'always') and $\mathcal{U}$ (read as 'until'). The use of this formalism 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 points T, $|w|_t$ denotes the truth value of the formula w at time t:

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

Informally this states that w is eventually true at time t if it is true at a time after t.

The other temporal operators can be defined in a similar fashion. A more complete introduction is omitted for the sake of brevity and the interested reader is directed to Prior (1967) and Rescher and Urquhart (1971).

The encapsulation of sequencing within definitions of temporal operators provides a tractable means of including timing information within specifications of complex safety-critical interfaces. For instance, temporal logic provides the designer with a means of describing global requirements, or axioms, of a system using the $\Box$ (read as 'always') operator:

$$\Box(interpret('start', off, startrun))$$

It is always the case that the input sequence *start* is interpreted as a command to start a run in the off state.

Delays during start-up procedures and internal system checks may prevent a designer from specifying the precise moment when a system will respond to operator commands. The introduction of the $\Diamond$ operator captures this uncertainty by specifying that a system should eventually handle user input:

$$eventualturnon \Leftarrow input('start') \wedge state(off) \wedge$$
$$interpret('start', off, startrun) \wedge \Diamond(effect(startrun, off, on))$$

A system is started if in the present interval *start* is input in the off state and this is interpreted as a command to start a run and eventually this has the effect of starting the system.

It is often vital to insure that some system responses occur within a stipulated interval. The $\bigcirc$ operator can be used to specify that a command must take effect immediately after it has been input from the user:

$$nextturnon \Leftarrow input('start') \wedge state(off) \wedge$$
$$interpret('start', off, startrun) \wedge \bigcirc(effect(startrun, off, on))$$

A system is started if in the present interval *start* is input in the off state and this is interpreted as a command to start a run and in the next interval this has the effect of starting the system.

This illustrates how imprecise temporal specifications can be constructed early in the development cycle using the $\Diamond$ operator. As a design is refined towards implementation it may be possible to guarantee more precise temporal properties specified using the $\bigcirc$ operator.

3 Applying Temporal Logic To Support Control

The following section illustrates the application of temporal logic by assessing the utility of structural decomposition and dialogue cycles proposed by human factors specialists.

3.1 Structural Decomposition

Buchanan and Bessant describe a complex control system in which seventy back-up dials and several VDU's simultaneously presented over one hundred and fifty process variables (Buchanan and Bessant, 1985). Complexity is not, however, simply related to the number of elements in a display. Henneman and Rouse (1984, 1986) show that although operators can adapt monitoring and scanning techniques to compensate for large amounts of information they cannot compensate for the presentation of inadequate or ill-structured data. Rasmussen and Lind (1981) and Bainbridge (1988) describe a number of structuring mechanisms intended to support the presentation of large amounts of information. One of the techniques advocated by these cognitive psychologists is the use of structural decomposition to represent the relationships between system components. Complex plant can be described in terms of its *parts*. For instance, a designer might specify that a coolant system is part of a casting system:

$$part(castingplant, coolantsystem).$$

This states that *coolantsystem* is part of *castingplant*.

We can decompose *coolantsystem*, in turn, to describe its component parts. For instance, it might include water jets, a number of coolant tanks and an inlet:

$$part(coolantsystem, waterjetA) \wedge part(coolantsystem, tankA) \wedge$$
$$part(coolantsystem, waterjetB) \wedge part(coolantsystem, tankB) \wedge$$
$$part(coolantsystem, inlet)$$

This states that $waterjetA$, $waterjetB$, $tankA$, $tankB$ and $inlet$ are all parts of the $coolantsystem$.

The resulting structure is illustrated by figure 3. This decomposition can be extended to form a hierarchy of parts ranging from nuts and bolts to an entire factory or site. The structural composition of control systems is not static. For example, diving operations from Occidental's Piper Alpha Oil Platform meant that automatic deluge equipment could not be operated. It could no longer be considered as *part* of the fire control systems:

> "(This) inhibited the operability of the safety system in an unnecessary and dangerous way." (Lord Cullen, 1990).

A parallel in the casting control example might be the loss of a tank from the coolant system:

$$\neg(part(coolantsystem, tankB)).$$

This states that $tankB$ is not a part of the $coolantsystem$.

The new structural decomposition of the coolant system is illustrated by figure 4. If a designer were restricted to the use of first order logic then they would be faced with a complex problem. Intuitively, this requirement could not be satisfied without an inconsistency because the previous decomposition of the coolant system stated that $tankB$ was a part of $coolantsystem$. This lack of monotonicity can be avoided by maintaining a database which records the current structure of the system. For instance, the loss of $tankB$ might force an update to the database to show that it is no longer part of the structure of the coolant system. This adds considerable complexity because designers must not only specify changes in structure but they must also check the consistency and coverage of their database. Logic programming languages provide automated support for this through built-in predicates, such as *assert* and *retract*. There are a number of technical problems which surround the use of such techniques (Moss 1986). Temporal logic provides an expressive and tractable alternative which avoids the additional complexity of forcing the designer to maintain a structural database. For example, a designer might require that an error in a tank has the effect of removing it from the cooling system:

$$(safeshutdown \Leftarrow part(coolantsystem, tankB) \wedge$$
$$\bigcirc(inputeffect('tankBerror', on, off) \wedge$$
$$view(off, coolantsystem) \wedge \neg(part(coolantsystem, tankB)))$$

This states that casting is safely shut down if $tankB$ is currently part of the coolant system and in the next interval input is received which indicates an error with this component and $tankB$ is not a part of the $coolantsystem$.

The temporal quantification provides the designer with a means of specifying that a component is part of a system in the present interval but not in the next. The use of the $\bigcirc$ operator avoids potential inconsistency and makes explicit the sequence in which structural change occurs.

3.2 Dialogue Cycles

Curry and Weiner (1980) describe how the auto-pilot of a B-707 disengaged without informing the crew. As a result the pilot could not predict the failure of his commands and the aircraft lost 30,000 feet before recovery was possible. One means of avoiding this danger would be to insure that a display never gives misleading state information, the auto-pilot should alert the crew to the fact that it had ceased to function! In the casting system this could be achieved by insuring that *errordisp* never displays the *on* or *off* states:

$$\Box(\neg(view(on, errordisp) \vee view(off, errordisp)))$$

It is never the case that *errordisp* is a display for the *on* state or the *off* state.

Unfortunately, it may be impossible to achieve this requirement for any but the simplest application. For instance, monitors may be sampled many times a second forcing subtle changes to the state of the system. The resources of both system and operator would be strained if the display were updated after every change in state. One solution is to hide state changes behind suitable abstractions of system behaviour. Some blast furnace control systems hide monitor readings by imposing delays between screen updates (Hoc 1987). Unfortunately, such information hiding can have a number of undesirable consequences which the interface designer must be aware of. For example, a display may eventually be presented for a number of system states which can occur before the screen is refreshed:

$$\Diamond(view(off, errordisp)) \wedge \Diamond(view(on, errordisp))$$

It is true that eventually *errordisp* is a view of the *off* state and eventually *errordisp* is a view of the *on* state.

This is a violation of our previous requirement because we know that for any formula w: $\Box(\neg(w)) \equiv \neg(\Diamond(w))$ (Manna and Pnueli 1981). The user may not be able to determine the whether the system is *on*, off or in *error* from the *errordisp* alone. If the user cannot determine whether the state of the system they may not be able to predict the effect of their commands. For instance, attempts to *start* a system may have very different effects if the system is off or in *error*:

$$\begin{aligned} &unpred('start', error, errordisp) \Leftrightarrow \\ &\quad \Diamond(view(off, errordisp) \wedge inputeffect('start', off, on)) \wedge \\ &\quad \Diamond(view(error, errordisp) \wedge inputeffect('start', error, error)) \end{aligned}$$

The effect of input *start* is unpredictable iff eventually *errordisp* is a view of the off state and input *start* has the effect of turning that system *on*. And eventually *errordisp* is a view of the *error* state and input *start* does not change the state of the system.

Although an operator may not know the exact state of the system between screen updates, it is important that they should be able to make certain predictions about the success or failure of commands. This can be achieved by emphasising cycles, or regularly occurring patterns, in interaction. Harrison, Roast and Wright (1989) describe dialogue cycles which are characterised by:

- an entry gate which marks the start of a dialogue;
- the redisplay of the entry gate which marks termination of the dialogue;
- the effect of the command (i.e., any associated change in the state of the system) only takes place once the dialogue has terminated.

The user may be confident that any command will be ineffective until the dialogue cycle has been completed. For instance, a designer might implement a casting control system in which an error may be resolved by raising the level of coolant to be sprayed onto the molten steel. A user might select a button to raise the level of coolant and thus change the state of the system from *error* to *on*. Once selected the appearance of the button would not change until the level of coolant had risen. The user may then predict that the command had not been successful until the button had resumed its original appearance. Figure 5 illustrates how this technique might be implemented. The inlet, described in the structural decomposition of the coolant system, is the button selected by an operator to raise the level of liquid in tanks A and B. The first diagram shows the image of the system at time **t** prior to the selection of *inlet*. The second diagram shows the image of the system at time **t+1** immediately after the selection has been made. The appearance of the inlet has changed to inform the user that their input is being handled by the system. The third diagram shows the image at time **t+2**, the appearance of inlet is the same as at **t+1** but the user is being informed of the effect of their command. The final diagram, **t+3** shows that the command has successfully terminated and the appearance of *inlet* has returned to that of time **t**. The user can be confident that the command has not been effective until *inlet* returns to its original appearance (i.e., it is no longer highlighted). In this example the gate of the cycle would include the appearance of the *inlet* during **t** and **t+3**. This presentation technique can be specified using the abstract model and the $\mathcal{U}$ (read as 'until') operator:

$$\forall p \in P, \exists g, d, d' \in D, \exists s, s' \in S : dialoguecycle(p, g, s, s') \Leftrightarrow$$
$$inputeffect(p, s, s') \wedge view(s, d) \wedge part(d, g) \wedge$$
$$(\neg state(s') \mathcal{U} (view(s', d') \wedge part(d', g)))$$

A dialogue cycle uses gate, g, to mark the effect of input p iff p has the effect of changing the state of the system from s to s' and the view of s is d and the *gate* is part of d and the state is not s' until the *gate* is again part of the display.

This abstract definition can be instantiated to express a dialogue requirement which might be posed for the casting control system:

$$dialoguecycle('raise', inlet, error, on)$$

A operator may observe the effect of input to raise the level of coolant in an error state by observing the *inlet* which is the gate of a dialogue cycle.

The success or failure of such cycles is determined by an operators ability to notice the gate, in this example the *inlet*, which marks the beginning and end of the dialogue. Human factors research has shown that inattention, distraction or fatigue might prevent users from noticing such elements of a display. This illustrates how specifications can include unwarranted assumptions about operator performance within a particular task or context of use. One means of avoiding such assumptions would be to base performance predictions upon models of the system operator. There are significant problems to be overcome before user models can be fully incorporated into interface design techniques. For instance, Rich (1983) acknowledges that individual differences may have to be considered if AI approaches to user modelling are to be

successful. Rouse (1981) assesses many of the problems facing the field of user modelling in psychology and concludes that computational complexity and inadequate heuristics currently limits their widespread application. A major international research effort is addressing these problems (Esprit Basic Research Action 3066). In anticipation of the results of this initiative it is extremely important that there be some alternative means of validating the formal design of interfaces to process control applications. For this reason we are concerned to adopt an iterative approach to design which exploits prototyping as a means of evaluating the products of formal design.

4 Prototyping

It is difficult to assess the impact of design decisions from their formal expression alone. Prototype implementations give a far better impression of what it would be like to interact with a system. Unfortunately, prototyping may introduce a considerable amount of complexity, including device handling and presentation details. Complexity increases the costs associated with maintaining multiple prototypes and this limits the investigation of alternative designs. With these problems in mind we have implemented Prelog, a tool for Presenting and REndering LOGic specifications of interactive systems (Johnson and Harrison 1990). Prelog combines a temporal logic interpreter with a structured graphics system (Pereira 1986) and high level device abstractions.

4.1 Executable Temporal Logic

Several research groups have attempted to provide an executable semantics for temporal logic specifications. For example, Hale (1987) and Moszkowski (1986) have implemented interpreters for the Tempura temporal logic programming language. Unfortunately, previous implementations have supported a limited range of input-output primitives. A further limitation is the imposition of left-right determinism. As a consequence valid temporal logic statements cannot be handled by Tempura:

$$display(I) \wedge (I \; is \; errordisp)$$

I is displayed in the present interval and I is the error display.

The interpreter scans from left to right and would fail because the identity of I is not known during the evaluation of $display(I)$. Such problems could be resolved by supporting backtracking or by enhancing the static analysis of programs using multiple-pass scanning techniques. Until these limitations are resolved, however, the interface designer must transform a specification to guarantee left-right determinism before they can be executed as Tempura programs.

The Prolog meta-interpreters Tokio (Aoyagi, Fujita and Moto-Oka 1986) and the Temporal Prolog of Sakuragawa (Sakuragawa, 1988) do not impose the determinism of Tempura. They do, however, restrict the designer to the Horn clause subset of logic which is supported by Prolog. This limitation was recognised and accepted when Tokio was incorporated into the Prelog system. If there are no temporal operators in a clause then Tokio will pass it directly on to Prolog for evaluation. Clauses which contain temporal operators are rewritten in first order logic and asserted over an appropriate interval. The interpreter maintains the fixed time points which otherwise might impose heavy burdens upon the designer. Execution progresses when all the clauses for a particular interval are satisfied (Fujita, Ishisone, Nakamura, Tanaka and Moto-Oka 1986). For example, a designer might specify that a system should display a system error followed by a display to show that the run was being aborted and lastly by a display that confirms the shut-down:

$$confusingshutdown :-$$
$$\bigcirc(display(offdisp)), display(errordisp), display(abortdisp).$$

A system shut down is displayed if in the next interval the $offdisp$ is presented and in the present interval $errordisp$ is displayed and $abortdisp$ is displayed.

This would achieve the desired sequence because the evaluation strategy of Tokio would fulfill the present requirements first in left-right order, $display(errordisp)$ then $display(abortdisp)$. Finally, the temporal evaluation would present the requirement for the next interval, $\bigcirc(display(offdisp))$. The sequencing might be made more explicit by avoiding any reliance upon the evaluation strategy of Prolog:

$$explicitshutdown :-$$
$$display(errordisp), \bigcirc(display(abortdisp), \bigcirc(display(offdisp))).$$

A system shut down is displayed if in the present interval $errordisp$ is presented next $abortdisp$ is displayed and finally $abortdisp$ is presented.

Tokio offers considerable benefits above Prolog as a platform for prototyping logic specifications. The dynamism of control forces the designer to constantly update records of the images currently displayed to the user. Schappo and Edmonds (1986) use meta-predicates, $assertg$ and $retractg$, to add and remove graphical objects from a Prolog database. This solution suffers from the same problems which surround the use of $assert$ and $retract$ mentioned in previous sections (Moss 1986). Prelog avoids these problems by using the higher order, temporal operators provided by Tokio to maintain the graphical database.

4.2 Structured Graphics

Previous sections have described how temporal logic provides a useful means of specifying the effects of structural change which may be presented to system operators. Decomposition of complex systems into their component parts supports the presentation of information at a variety of levels of granularity. At the top of the hierarchy designers might present global details about the state of a site or factory, at the bottom of the hierarchy there may be data about low-level process components. In order to implement such a structure within a prototype it must be supported by graphical primitives which possess the representational attributes presented to the user. For instance, $inlet$ can be described in terms of lines:

$$simpleimage(inlet) :-$$
$$primitive(inlet), line(inlet, 0.1, 0.3, 0.6, 0.3),$$
$$line(inlet, 0.1, 0.2, 0.6, 0.2), line(inlet, 0.6, 0.2, 0.6, 0.8).$$

$Inlet$ has a simple graphical appearance if it is a primitive graphical image which has no component $parts$ and the graphical appearance of $inlet$ includes a line from X-Y coordinates (0.1,0.2) to (0.6,0.2). And the graphical appearance of $inlet$ includes a line from X-Y coordinates (0.6,0.2) to (0.6,0.8) and another from (0.1,0.3) to (0.6,0.3).

Realistic prototyping tools for control systems must be capable of supporting complex graphical structures and not just primitive lines. A user selecting an inlet icon does not expect a particular line of the image to be selected. Similarly they should not be forced to select a particular line in order to select a region, such as that enclosed by the lines of the inlet icon. Prelog has, therefore, been developed to provide

access to the Presenter screen management system.

Presenter supports the decomposition of complex images into a number of regions. For example, a screen may be described in terms of a background region, a text region and an inlet region. The inlet icon might be further decomposed into regions containing graphical objects such as squares or circles. Figure 6 illustrates the graphical decomposition for part of the image of the coolant system. Each region has properties, such as size and position, attributes, such as font or colour, and a behaviour, such as whether it is selectable or not. This approach isolates low-level graphical representations, such as bitmaps or raster operations, within Presenter. This prevents them from threatening the tractability of the high-level temporal specifications of interface behaviour constructed by the designer. Graphical decomposition is appropriate for Prelog because it is analogous to the structural decomposition advocated by the human factors analysis of control tasks. For instance, the image of *inlet* might be described as follows:

$$\begin{aligned}&compleximage(inlet) :- \\ &\quad part(screen, coolantdisplay), part(coolantdisplay, inlet), \\ &\quad region(inlet), dimension(inlet, 0.2, 0.2), \\ &\quad position(inlet, 0.4, 0.2), pattern(inlet, blank).\end{aligned}$$

Inlet has a complex graphical appearance if *coolantdisplay* is part of the screen image currently displayed and *inlet* is part of *coolantdisplay*. And *inlet* is a primitive region of dimension 0.2 by 0.2 units at (X,Y) position (0.4, 0.2) and has a blank pattern.

Changes to the graphical composition or appearance of a prototype are elegantly expressed using temporal logic in a similar fashion to that described for changes to the structural composition:

$$\begin{aligned}&newimage(inlet) :- compleximage(inlet), \\ &\quad \bigcirc(pattern(inlet, dark)).\end{aligned}$$

Inlet changes its appearance if in the present interval its appearance is the *compleximage* described in previous clauses and in the next interval its pattern is now dark (i.e., shaded).

This clause describes how a designer might implement the highlighting which indicates the *gate* of the dialogue cycle described in previous sections.

4.3 Device Abstractions

There are computational and representational problems with handling devices directly from within Prolog interpreters (Clark, McCabe, Johns and Spenser 1987, McCabe 1987). Prelog resolves these problem by representing physical devices, such as mice or keyboards, by the events they generate. Events can be associated with the regions which form part of the graphical decomposition of control system components. This is easily achieved because Presenter provides applications with input in the form of events associated with particular regions. An important benefit of this approach is that it supports implementation without commitment to particular devices and hence need not sacrifice a primary aim of abstract specifications. Designers can produce a number of prototypes, using different interaction devices, without sacrificing the tractability of high level designs. For example, a designer might present an inlet icon which must be selected to increase the level of coolant in a tank, the system operator could perform this selection using a mouse, touch screen or cursor keys. All of these physical devices would generate the same selection event. This device independence reduces the costs and complexity associated with maintaining prototypes which

exploit a variety of different input devices. We can incorporate events directly into a specification:

$$presraiselevel2 : -inputeffect(select(inlet), error, on),$$
$$view(error, coolantdisplay), part(coolantdisplay, inlet),$$
$$(not(state(on))\mathcal{U}(view(on, coolantdisplay),$$
$$part(coolantdisplay, inlet))).$$

A user may predict the effect of raising the level of coolant in an error state iff the effect of selecting *inlet* is to transform a system from the *error* to the *on* state and *coolantdisplay* is presented during the *error* state and *inlet* is part of *coolantsystem*. And the state of the system is not *on* until *coolantdisplay* is presented for that state and the *inlet* is again part of that display.

In this example the *inlet* of the coolant system is the subject of the generic *select* event. The specification would not need to be changed if a prototype used a tracker ball or a mouse to generate it. The gate of the dialogue cycle is marked by *(view(on, coolantdisplay), part(coolantdisplay, inlet))* and the $\mathcal{U}$ operator is implemented using the temporal interpreter within Prelog. To summarise, temporal logic interpreters provide a means of extending formal specification techniques to support the implementation of prototypes. These provide a cost-effective means of evaluating the presentation techniques proposed by human factors research prior to full implementation. The use of graphical decomposition and device abstraction supports this implementation without incurring undue device dependency early in the development cycle.

4.4 Implementation

Prelog is designed around the object or server model. This architecture describes how shared resources can be accessed by a number of clients through a central server. In the current implementation, clients are assumed to be Presenter applications and the resource is the temporal logic specification running under Tokio. This design provides great flexibility, Prelog is readily adapted to run over a distributed system. This has advantages for user testing, the designer can monitor the execution of a prototype on a remote machine. It also provides a powerful means of prototyping computer supported cooperative work (CSCW) systems. Temporal logic is well suited to reasoning about the deadlock and contention which may arise when systems provide concurrent access to multiple users of common resources (Johnson 1991). Figure 7 illustrates how the Prelog architecture incorporates Tokio, Presenter and Prolog.

5 Future Work

Further work intends to extend the application of temporal logic in a number of ways:

- The tools and techniques already developed will be tested against a number of extended examples. These might include selected sub-systems from power generation or chemical process control. This investigation will also be extended to assess the utility of alternative presentation techniques to support the operation of dynamic systems. For example, browsing strategies such as fish-eye displays provide users with a means of focusing attention within graphical or structural hierarchies (Furnas 1986). These techniques have not previously been extended to present structural change. It is hypothesised that temporal logic and the graphical resources of Prelog will provide a powerful means of achieving this;
- The integration of modelling techniques from both cognitive psychology and computer science remains a thorny problems and it is hoped that further work will address this. It is interesting to

note that temporal logic is currently being assessed as a means of extending cognitive modelling techniques (Anderson, 1990). Such work will continue the integration of human factors and formal design;

- Research should be undertaken to investigate suitable extensions to the logical formalisms described in this paper. For instance, the incorporation of real-time into interval temporal logic is a subject which requires some consideration. It is hoped to further investigate the conflicting temporal properties of prototypes and specifications. For instance, $\Diamond$ properties are appropriate for the early stages of specification. They avoid commitment to a particular interval and can be satisfied in the present interval or in the next interval or in the interval after that etc. They may, however, jeopardise safety by postponing error displays in an implementation until they are eventually presented long after the system had failed (Johnson 1990a);
- The presentation resources of the structured graphics system should be extended and enhanced. Colour plays an important part in many current control systems and it must be incorporated into Prelog.

This agenda must be completed before Prelog can adequately bridge the divide between 'laboratory bench' and 'shop floor' identified in the introduction.

6 Conclusion

This paper has argued that the lack of adequate tools has impaired the transfer of research in cognitive and perceptual psychology into commercial interface design. Formal methods have been proposed as a solution capable of bridging this divide between systems and human factors engineering. They provide a precise and concise means of both combating complexity for the designer and of expressing the techniques intended to resolve complexity for system operators. Unfortunately, formal methods have failed to capture the temporal properties and device characteristics which may determine the usability of an interface. These deficiencies have been overcome by extending temporal reasoning from AI research and concurrency theory to support the specification and prototyping of interactive control systems. In particular, temporal logic has been proposed as a powerful and expressive means of analysing usability requirements. Executable temporal logics can be employed to support prototyping and this provides a means of validating the techniques advocated by human factors specialists. Further work intends to investigate the utility of this approach for a variety of complex examples. It is hoped these techniques will support the realisation of human factors research within future generations of increasingly complex and safety-critical control systems.

7 Acknowledgements

Thanks are due to the referees whose helpful comments and criticisms guided the evolution of this paper. Particular thanks are due to the editor who provided valuable pointers to related work. Discussions with Gregory Abowd, Chris Roast and the members of the Human Computer Interaction group at the University of York inspired many of the ideas presented in this work. This research has been supported by SERC grant number 88503497, a CASE award funded by British Telecom and by the 1990 Gibbs-Plessey travel scholarship.

8 References

Alexander, H. (1987), Executable specifications as an aid to dialogue design. In Bullinger and Shackel, editors, *Interact '87*. North Holland: Elsevier Scientific.

Alty, J.L., Elzer, P., Holst, O., Johannsen, G. and Savory, S. (1986), Technical Report AMU8603/01S-ESPRIT:P600, The Scottish HCI Centre, Strathclyde University, United Kingdom.

Anderson, H. (1990), On Temporal Thought. Personal Communication, The Risø National Laboratory, DK-4000 Roskilde, Denmark.

Aoyagi, T., Fujita, M. and Moto-Oka, T. (1986), Temporal logic programming language - Tokio - programming in Tokio. In E. Wada, editor, *Logic Programming '85*. Berlin, FDR: Springer Verlag.

Bainbridge, L. (1987), Ironies Of Automation, In J. Rasmussen, K. Duncan and J. Leplat, editors, *New Technology And Human Error*, pages 271 - 283. New York, United States of America: J. Wiley and Sons.

Bainbridge, L. (1988), Types Of Representation, In L.P. Goodstein, H.B. Anderson and S.E. Olsen, editors, *Task, Errors And Mental Models*, pages 70 - 91. London, United Kingdom: Taylor and Francis.

Beltracchi, L. (1989), Energy, Mass, Model-Based Displays and Memory Recall. *IEEE Transactions On Nuclear Science*, 36, 3, 1367-1382.

Bolour, A., Anderson, T.L., De Keyser, L.J. and Wong, H.K.T. (1982), The Role Of Time In Information Processing: A Survey. *ACM SIGMOD*, 12, 3, 27-50.

Buchanan, D.A. and Bessant, J. (1985), Failure, uncertainty and control : The role of the operator in a computer integrated production system. *The Journal Of Management Studies*, 22, 3, 292-308.

Card, S.K., Mackinlay, J.D. and Robertson, G.G. (1990), The Design Space Of Input Devices. In J.C. Chew and J. Whiteside, editors, *Proceedings Of CHI '90*, pages 117-124, Reading, MA, United States Of America: Addison Wesley.

Clark, K.L. (1981), The Synthesis And Verification Of Logic Programs, Technical Report 81/36, Department of Computing, Imperial College Of Science And Technology, London, United Kingdom.

Clark, K.L., McCabe, F., Johns, N. and Spenser, C. (1987), *L.P.A. MacProlog Reference Manual*. London, United Kingdom: L.P.A. Ltd.

Clegg, C. and Wall, T.D. (1987), Managing factory automation. In F. Blackler and D. Obourne (editors), Information Technology And People, pages 45-64, Leicester, United Kingdom: The British Psychological Society.

Lord Cullen. (1990) cited in *The Times*, 13 November, page 5.

Dix, A.J, (1987), Formal Methods And Interactive Systems: Principles And Practice. DPhil Thesis, Department of Computer Science, University of York, United Kingdom.

Forbus, K.D. (1987), Interpreting observations of physical systems. *IEEE Transactions On Systems, Man And Cybernetics*, SMC-17, 3, 350-359.

Fujita, M., Ishisone, M., Nakamura, H., Tanaka, H. and Moto-Oka, T. (1986), Using The Temporal Logic Programming Language TOKIO For Algorithm Description And Automatic CMOS Gate Array Synthesis. In E. Wada, editor, *Logic Programming '85*, pages 246 - 255, Berlin, FDR: Springer Verlag.

Furnas, G.W. (1986), Generalised fish-eye views. In M. Mantei and P. Orbeton, editors, *Human Factors In Computing Systems - CHI '86 Conference Proceedings*. Boston, United States Of America: A.C.M..

Gabbay, D.M. (1987), Modal And Temporal Logic Programming. In A. Galton, editor, *Temporal Logics And Their Applications*, pages 197 - 223. London, United Kingdom: Academic Press.

Gaines, B.R. (1988), A Conceptual Framework For Person-Computer Interaction In Complex Systems, *IEEE Transactions On Systems, Man And Cybernetics*, SMC-18, 4, 532-541.

Green, A.E. (1988), Human factors in industrial risk assessment - some early work. In L.P. Goodstein, H.B. Anderson, and S.E. Olsen, editors, *Task, Errors And Mental Models*, pages 193-208. London, United Kingdom: Taylor and Francis.

Hale, R. (1987), Temporal Logic Programming. In A. Galton, editors, *Temporal Logics And Their Applications*, pages 91-119. London, United Kingdom: Academic Press.

Harrison, M.D. and Dix, A.J. (1990), A state model of direct manipulation in interactive systems. In M.D. Harrison and H. Thimbleby, editors, *Formal Methods In Human Computer Interaction*, pages 131-153. Cambridge, United Kingdom: Cambridge University Press.

Harrison, M.D., Roast, C.R. and Wright, P.C. (1989), Complimentary methods for the iterative design of interactive systems. In G. Salvendy and M.J. Smith, editors, *Designing And Using Human-Computer Interfaces And Knowledge Based Systems*, pages 651-658. North Holland: Elsevier Scientific.

Hawkins, J.S., Reisning, J.M., Lizza, G.D. and Beachy, K.A. (1983). In Proceedings Of The 27th Annual Meeting Of The Human Factors Society, pages 970-972. Santa Monica, United States Of America: Human Factors Society.

Henneman, R.L. and Rouse, W.B. (1984), Human Performance In Monitoring And Controlling Hierarchical Large-Scale Systems, *IEEE Transactions On Systems, Man And Cybernetics*, SMC-14, 2, 184-191.

Henneman, R.L. and Rouse, W.B. (1986), On Measuring The Complexity Of Monitoring And Controlling Large-Scale Systems, *IEEE Transactions On Systems, Man And Cybernetics*, SMC-16, 2, 193-207.

Hoc, J.-M. (1987), Analysis of cognitive activities in process control for the design of computer aids - an example the control of a blast furnace. In H.-J. Bullinger and B. Shackel, editors, *Interact '87*, pages 257-262. North Holland: Elsevier Scientific.

Johnson, C.W. (1990), Applying high level models to the design of interactive process control systems. Technical report 90/3, Faculty of Mathematics, The Open University, Milton Keynes, United Kingdom.

Johnson, C.W. (1990a), Using temporal logic to prototype interactive systems. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel, editors, *Interact '90*, pages 1019-1021. North Holland: Elsevier Scientific.

Johnson, C.W. (1991), Applying temporal logic to support the design of concurrent multi-user interfaces. To appear in D. Diaper and N. Hammond, editors, HCI'91: Usability Now. British Computer Society: London.

Johnson, C.W. and Harrison, M.D. (1990), PRELOG - a system for presenting and rendering logic specifications of interactive systems. In C.E. Vandoni and D.A. Duce, editors, *Eurographics '90*, pages 469-480. North Holland: Elsevier Scientific.

Johnson, C.W. and Harrison, M.D. (1991), Declarative graphics and dynamic interaction. To appear in J. Schonut and L. Kjelldahl, editors, *Eurographics '91*. North Holland: Elsevier Scientific.

De Keyser, V. (1987), The structuring of knowledge of operators in continuous processes : A case study of a continuous casting-plant start up. In J. Rasmussen, K. Duncan, and J. Leplat, editors, *New Technology And Human Error*, pages 247-259. New York, United States Of America: J. Wiley And Sons.

Kowalski, R. (1979), Algorithm = Logic + Control. *Communications Of The A.C.M.*, 22, 7, 424-426.

Kuhmann, W. (1989), Stress inducing properties of system response times. *Ergonomics*, 32, 3, 271-280.

Langefors, B. and Sundgren, B. (1975), *Information Systems Architecture*. New York, United States Of America: Charter Press.

McCabe, F.G. (1987), Denotational graphics. Technical report, Department of Computing, Imperial College Of Science And Technology, London, United Kingdom.

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

Moray, N. and Rotenberg, I. (1989), Fault Management In Process Control: Eye Movements And Action. *Ergonomics*, 32, 11, 1319-1342.

Moss, C. (1986), Cut and paste : Defining the impure primitives of Prolog. In E. Shapiro, editor, *Third International Conference On Logic Programming*, pages 686-694. Berlin, FDR: Springer-Verlag.

Moszkowski, B. (1986), *Executing Temporal Logic Programs*. Cambridge, United Kingdom: Cambridge University Press.

Pereira, F.C.N. (1986), Can drawing be liberated from the Von Neumann style? In M. Van Caneghan and D. H. D. Warren, editors, *Logic Programming And Its Application*, pages 175-187. Norwood, New Jersey, United States of America: Ablex Publishing.

Prior, A. (1967), *Past, Present, Future*. Oxford, United Kingdom: Oxford University Press.

Rasmussen, J. and Lind, M. (1981), Coping With Complexity. Technical Report Risø-M-2293, The Risø National Laboratory, DK-4000 Roskilde, Denmark.

Rasmussen, J. (1983), Skills, Rules And Knowledge; Signals, Signs and Symbols and Other Distinctions In Human Performance Models. *IEEE Transactions On Systems, Man And Cybernetics*, SMC-13, 3, 257-266.

Reason, J. (1987), The psychology of mistakes: a brief review of planning failures. In J. Rasmussen, K. Duncan and J. Leplat (editors), *New Technology And Human Error*, pages 45-53. Chichester, United Kingdom: John Wiley And Sons

Rescher, N. and Urquhart A. (1971), *Temporal Logic*. Berlin, FDR: Springer Verlag.

Rich, E. (1983), Users are individuals: individualising user models. *International Journal of Man-Machine Studies*, 18, 199-214.

Rouse, W.B. (1981), Human computer interaction in the control of dynamic systems. *ACM Computing Surveys*, 13, 1, 71-99.

Sakuragawa, T. (1987), Temporal Prolog - A Programming Language Based On Temporal Logic, *Computer Software*, 4, 3, 15-27 (in Japanese).

Sakuragawa, T. (1988), RACCO: A Modal-Logic Programming Language For Writing Models Of Real-Time Process-Control Systems, *Computer Software*, 5, 3, 22-33 (in Japanese).

Schappo, A. and Edmonds, E.A. (1986), Support for tentative design : Incorporating the screen, as a graphical object, into Prolog. *Int. Journal Of Man Machine Studies*, 24, 601-609.

Singleton, W.T. (1973), Theoretical Approaches To Human Error, *Ergonomics*, 16, 727-737.

Stephens, M. (1980), *Three Mile Island*. London, United Kingdom: Junction Books.

Suffrin, B. and He, J. (1990), Specification, analysis and refinement of interactive processes. In M.D. Harrison and H. Thimbleby, editors, *Formal Methods In Human Computer Interaction*, pages 153-199. Cambridge, United Kingdom: Cambridge University Press.

Wiener, E.L. and Curry, R.E. (1980), Flight Deck Automation : Promises And Problems, *Ergonomics*, 23, 10, 995-1011.

Woods, D.D. (1988), Coping with complexity: the psychology of human behaviour in complex systems. In L.P. Goodstein, H.B. Anderson and S.E. Olsen (editors), *Task, Errors and Mental Models* pages 128-148. London, United Kingdom: Taylor and Francis.

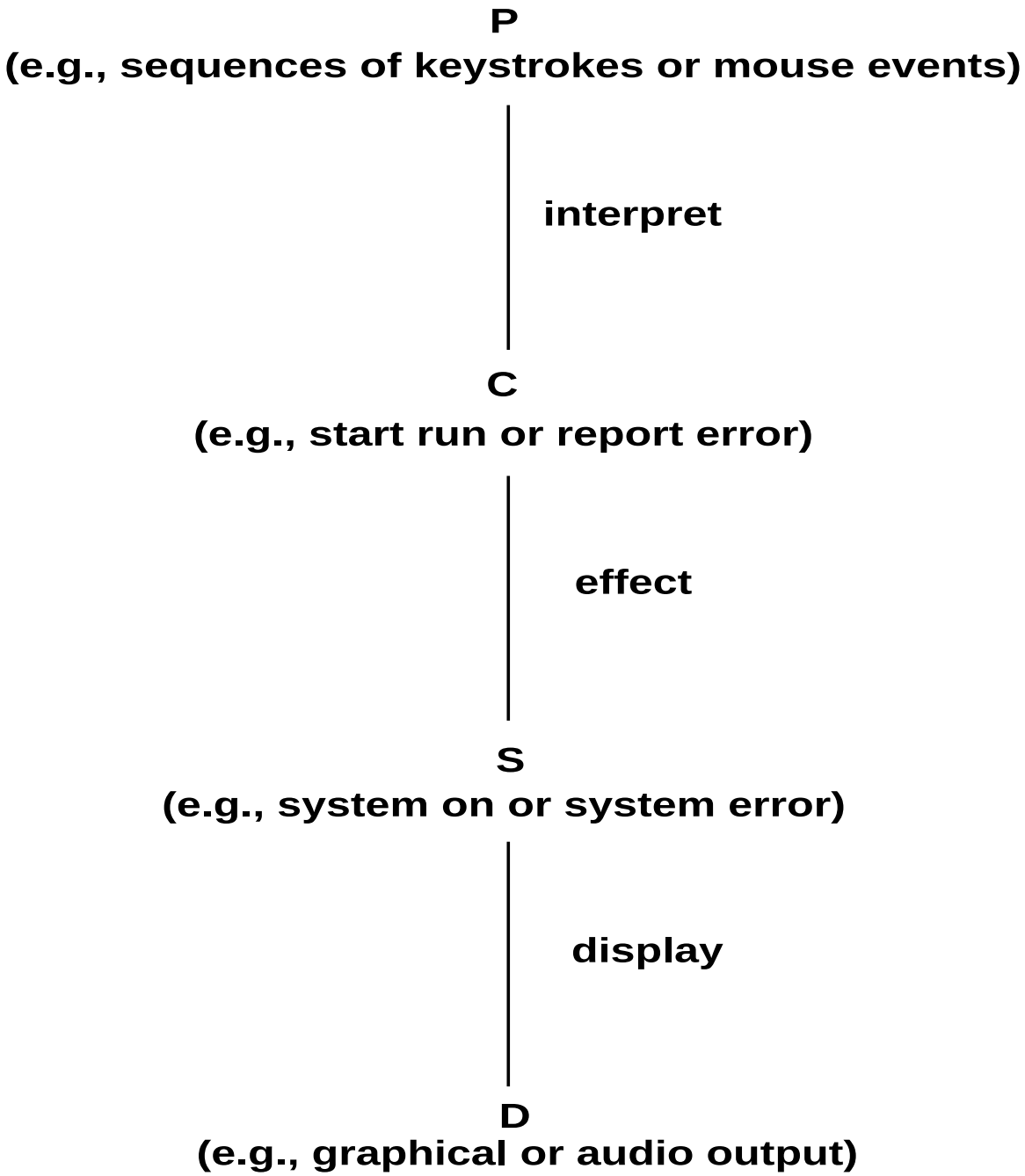


Figure 1: A High Level Model Of Interaction.

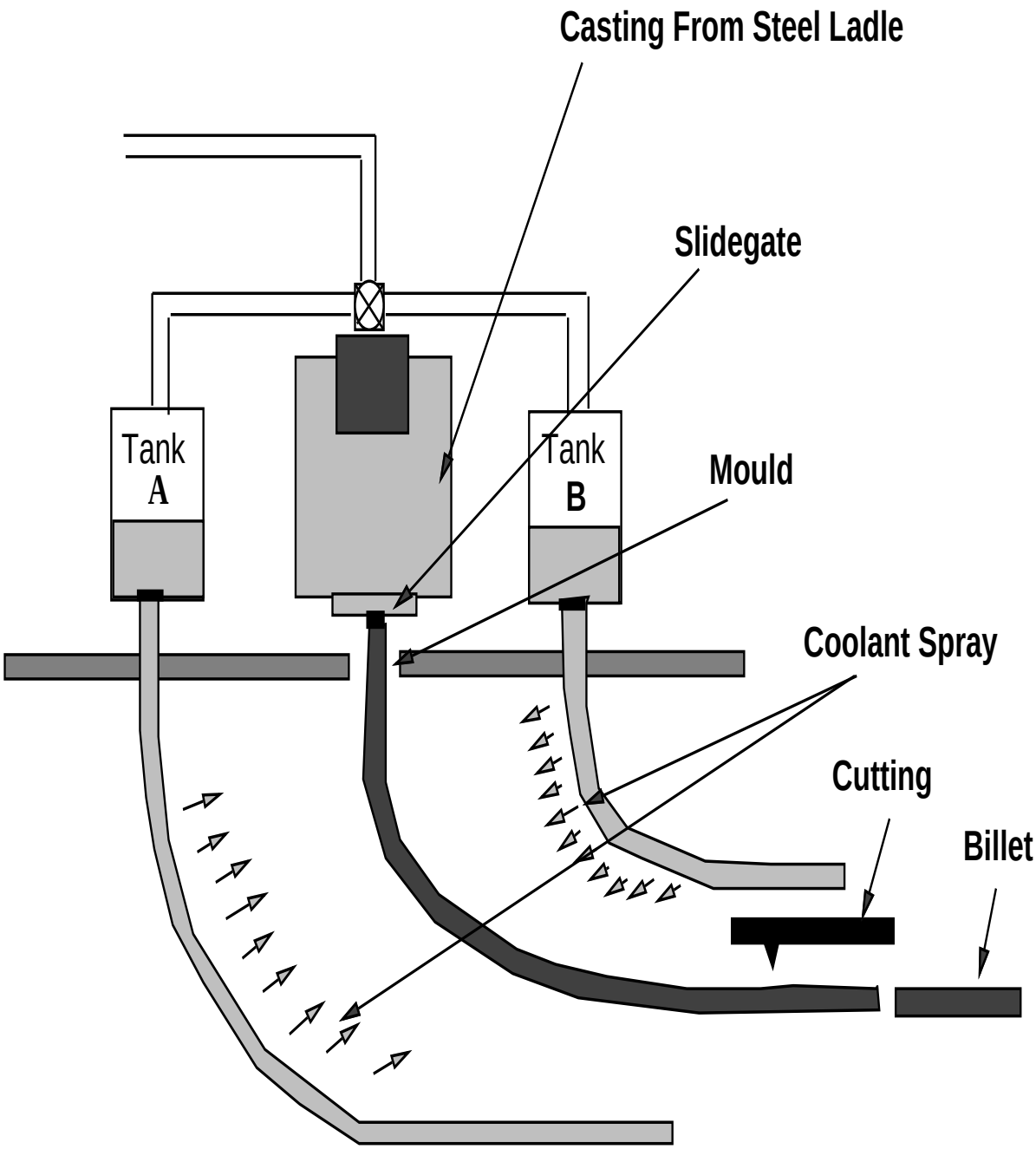
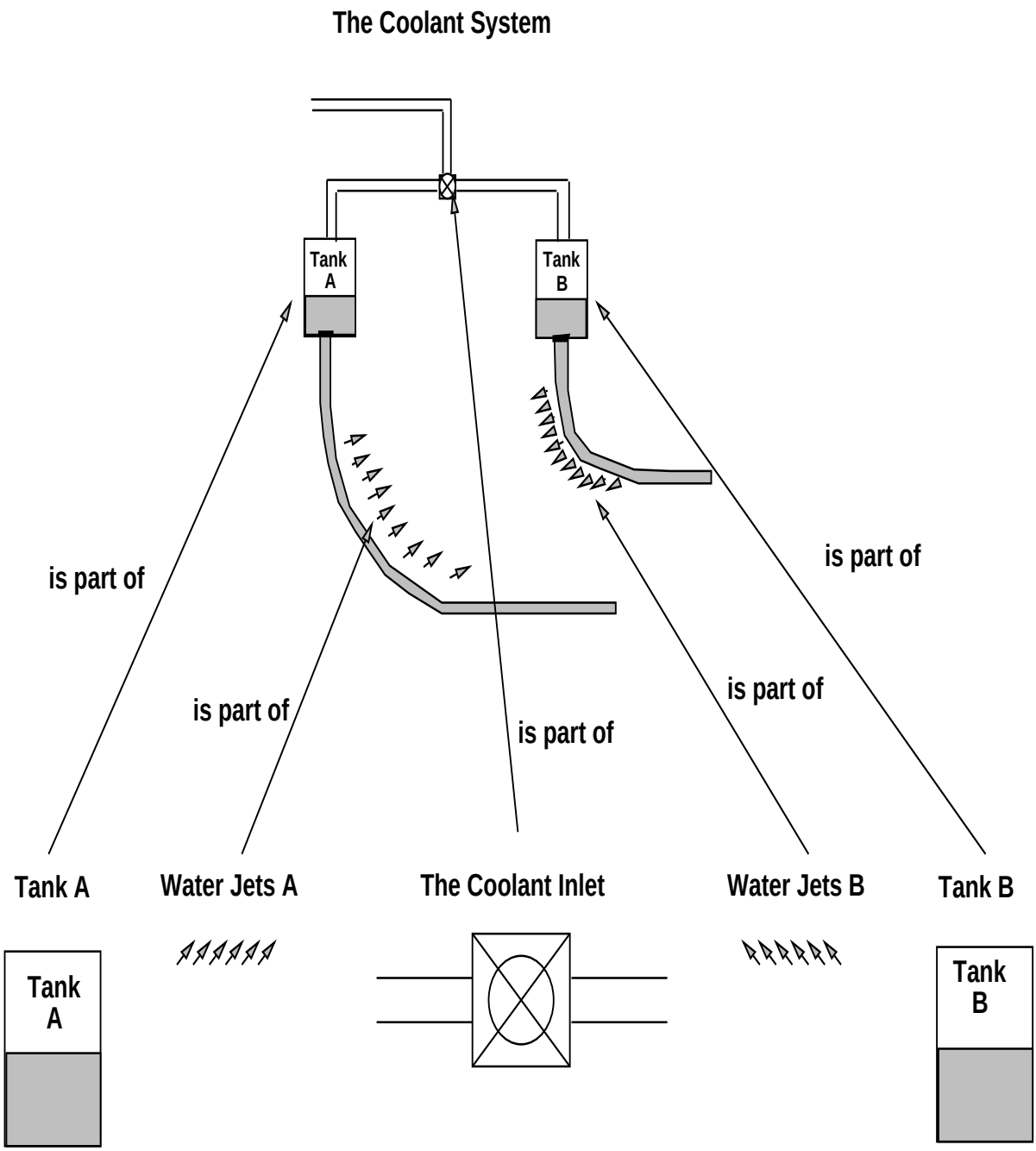


Figure 2: The Continuous Casting Process


The Coolant System
Tank A
Tank B
is part of
is part of
is part of
is part of
is part of
Tank A
Water Jets A
The Coolant Inlet
Water Jets B
Tank B
Tank A
Tank B

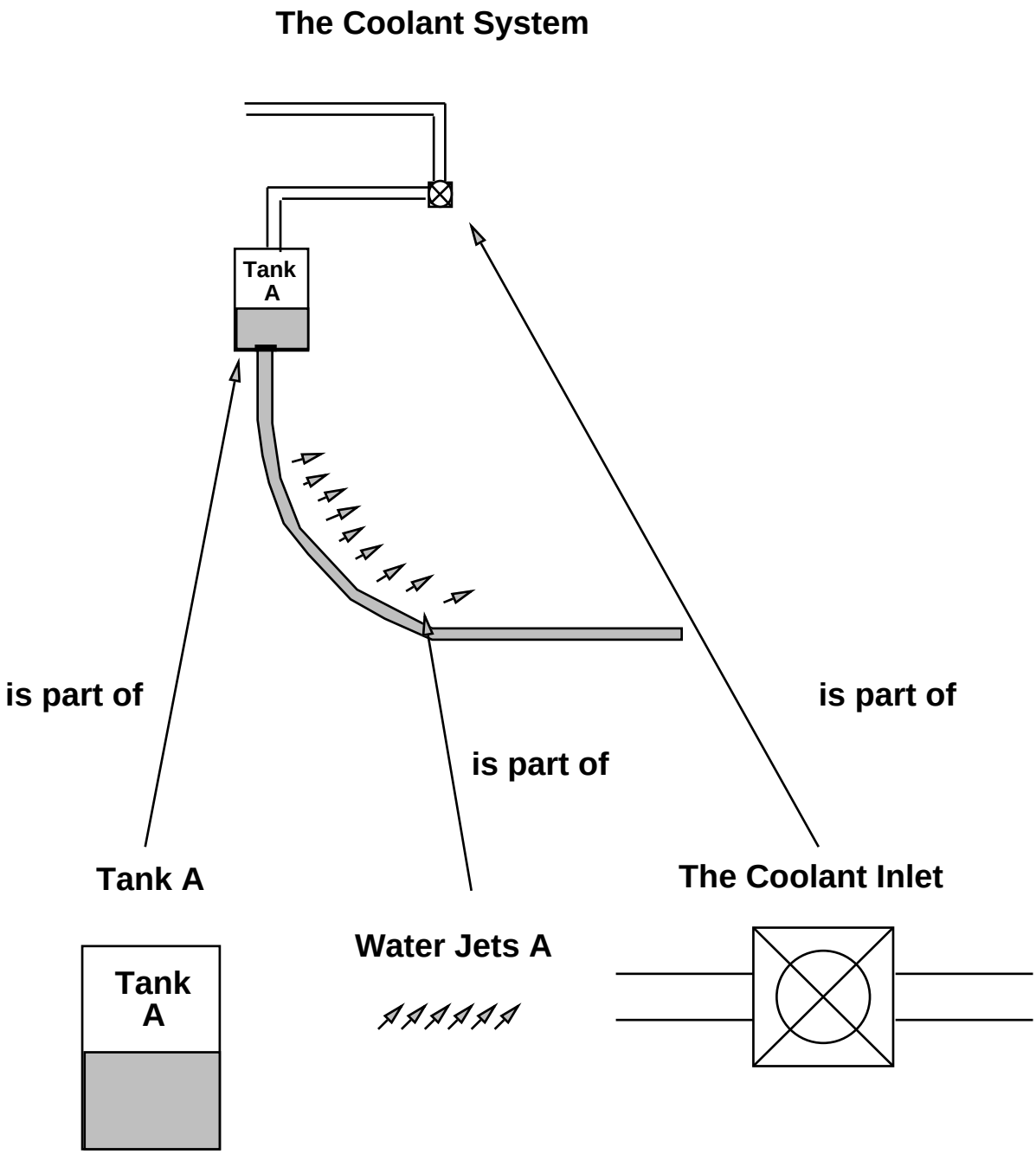


Figure 3: Structural Decomposition Of *Coolantsystem*.

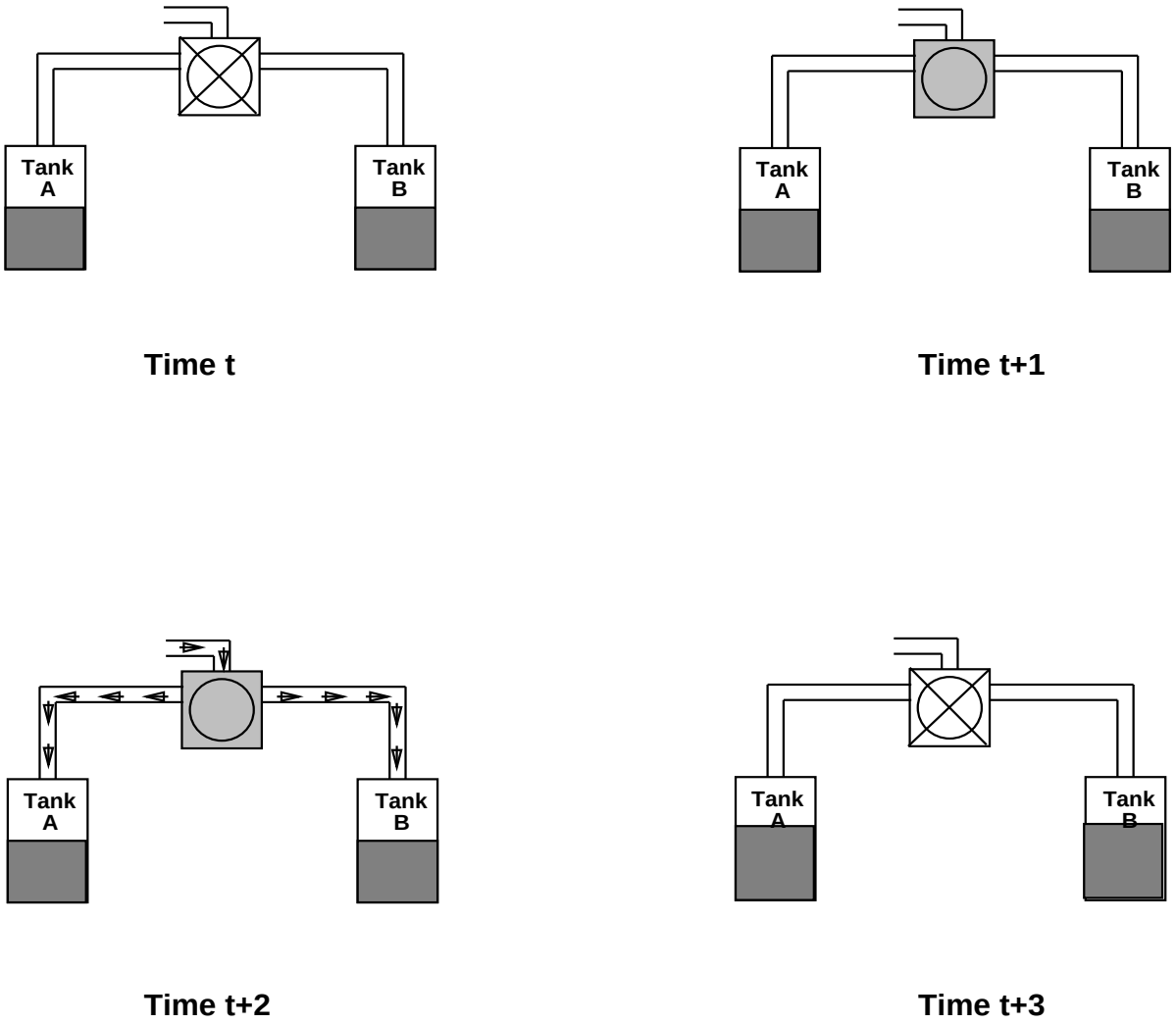


Figure 4: An Example Of A Dialogue Cycle.

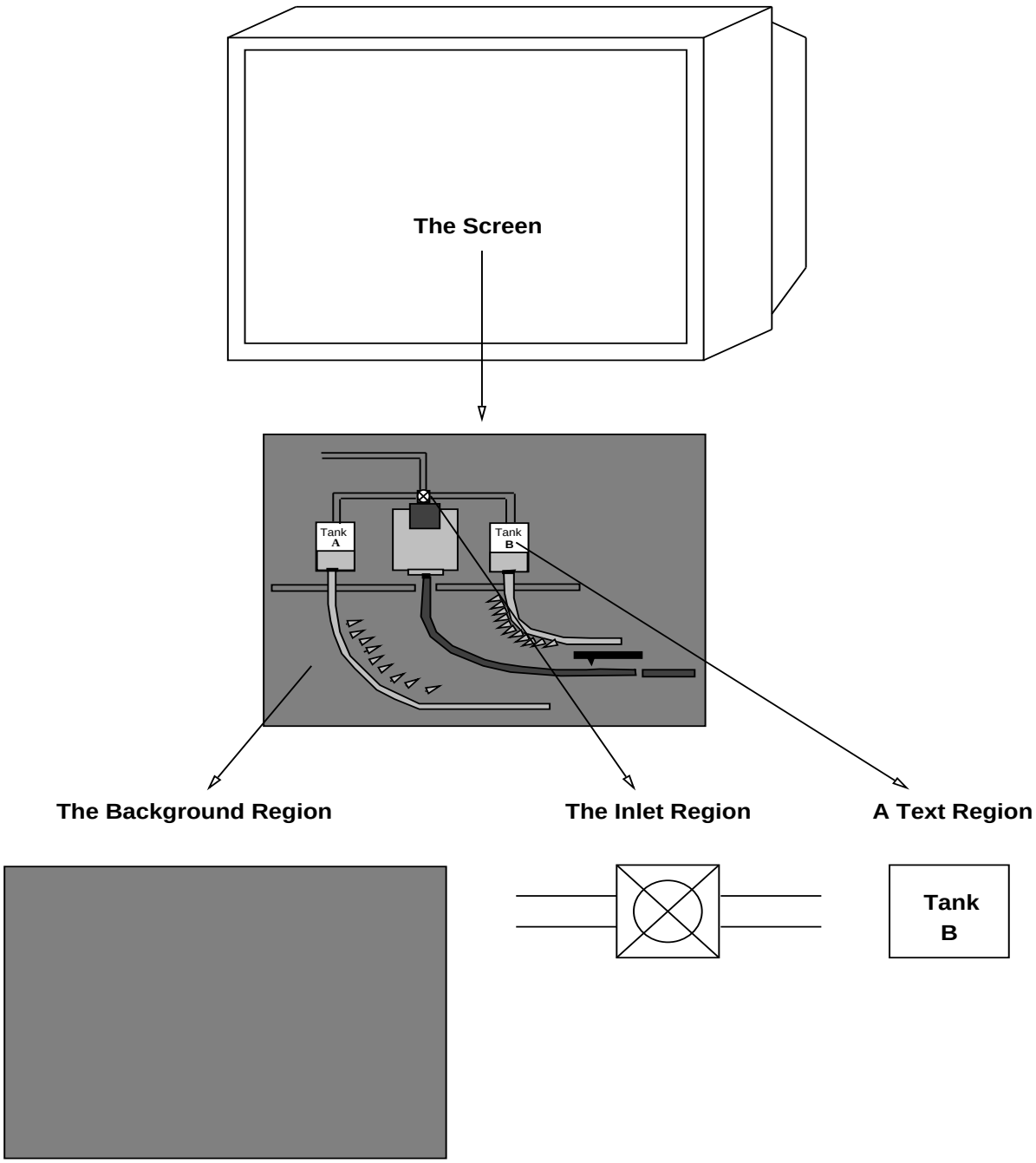


Figure 5: Graphical Decomposition Of Part Of The *Castingplant*.

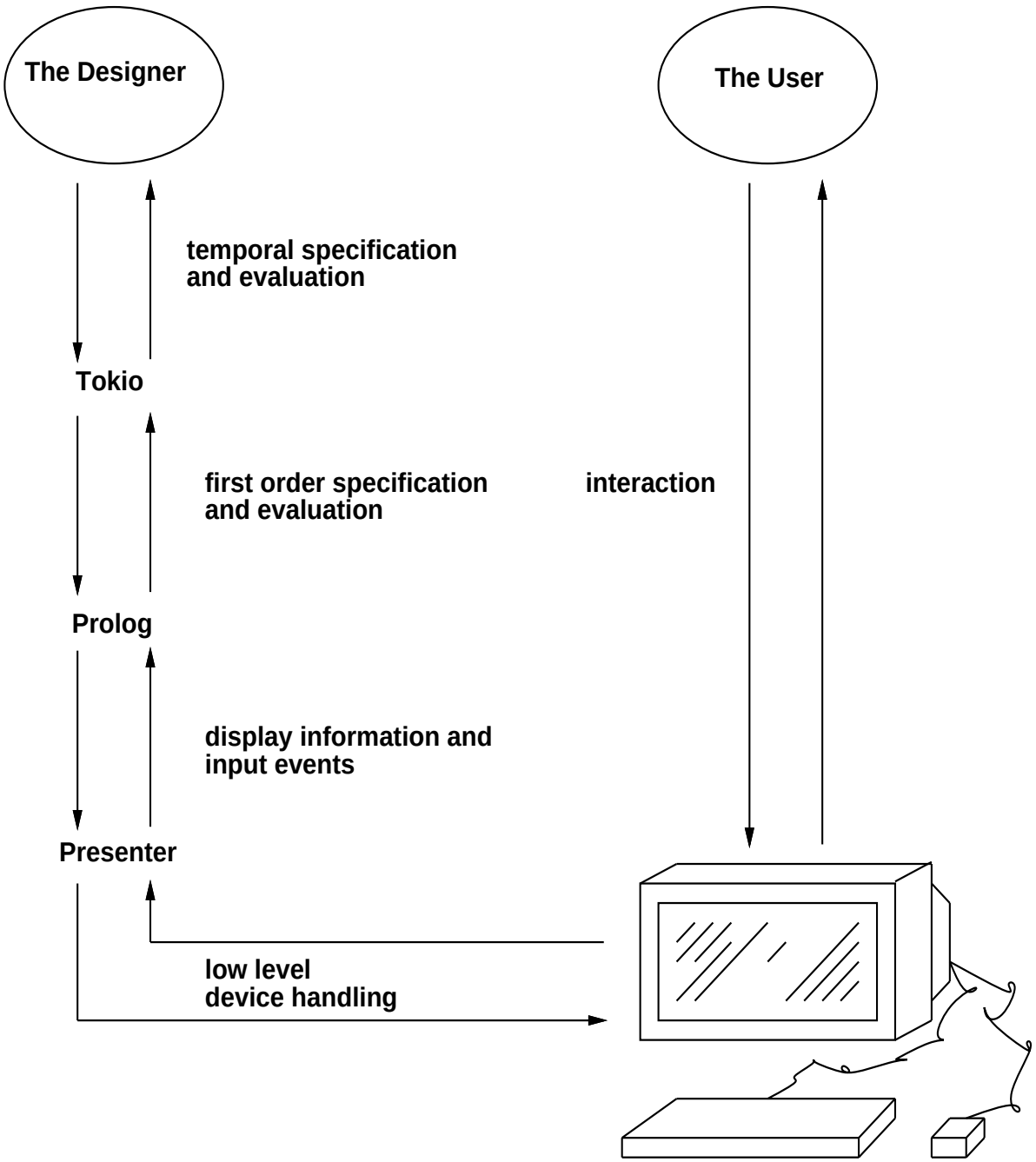
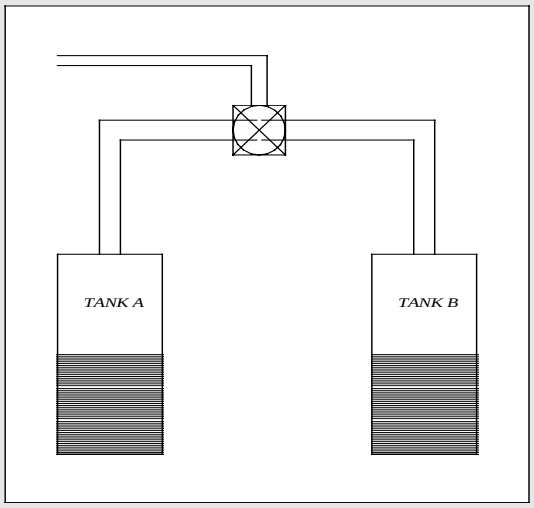
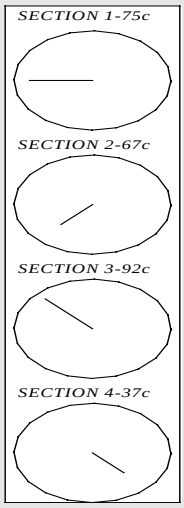
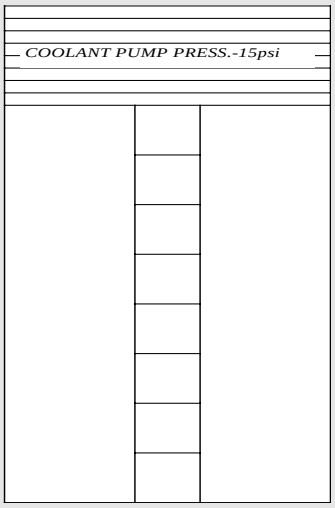


Figure 6: The Prelog Architecture.

SYSTEM -HELP-	REFINE -VIEW-	EXPAND -VIEW-	FAULT -REPORT-	MAILING -IN/OUT-	REVIEW -HISTORY	PREVIEW -FUTURE-	RESET -PARAM-	ALTER -PARAM-	ERROR -ABORT-







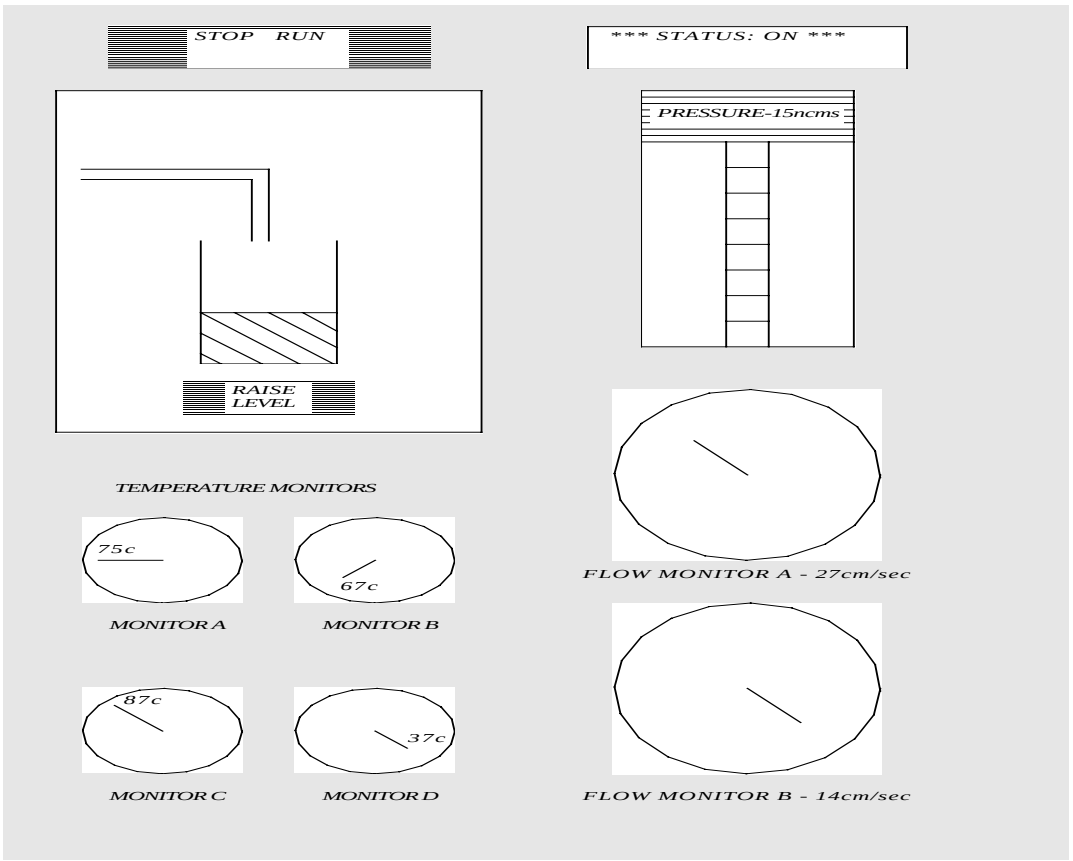


Figure 7: Prelog Prototypes Of The Casting Control System.