A Formal Approach To The Presentation Of CSCW Systems

C.W. Johnson
Department of Computer Science, University of York,
Heslington, York, United Kingdom, YO1 5DD.
E-mail: johnson@minster.york.ac.uk,
Telephone: (0904) 433376, Fax: (0904) 432767.

ABSTRACT

It is a non-trivial task to develop appropriate presentation strategies for Computer Supported Cooperative Work (CSCW) systems. Some applications, such as communications environments, must be presented to all the members of a group. Other information is only relevant for those users who are engaged in particular activities. Representing a design in terms of bitmaps and device primitives can obscure such requirements. This paper argues that formal, mathematically based, specification techniques can represent images that are distributed amongst multiple, concurrent operators. A limitation of this approach is that formal specifications provide little impression of what it would be like to interact with a CSCW system. The Prelog prototyping tool has been developed to overcome this limitation. Prelog can be used to derive prototypes from logic specifications of multi-user applications. Prelog is intended to support the early stages of development when it may not be possible to analyse systems within their eventual working context. It also supports the later stages of development because Prelog displays can be incorporated into final implementations using either the C or Ada programming languages.

Keywords: *CSCW; formal methods; Prelog; graphics programming.*

1 Introduction

Figure 1 presents Williges, Williges and Elkerton's three-stage model of interface development [26]. Initial design is intended to elicit and record the requirements which are to be imposed upon an interface. Formative evaluation validates those requirements. Summative evaluation ensures that an implementation satisfies its requirements. It is possible to use this model to identify a number of problems which arise during the development of CSCW systems. Locking and synchronisation techniques impose additional burdens which are not apparent during the initial development of single-user applications. The CSCW literature presents a range of mechanisms which can be used to resolve these problems [22, 8]. In contrast, comparatively little work has been done to support the development of multi-user displays. It is not clear what notations or techniques might be used to support the initial design of presentation techniques for CSCW systems. Natural language descriptions of concurrent displays can quickly become intractable. This is a significant problem; it is a non-trivial task to develop appropriate presentation strategies for multi-user applications. If necessary information is hidden from some users then many of the benefits of groupware may be lost. Other information is only relevant for those members of a group who are engaged in particular tasks. A further problem is that designers must validate presentation requirements during the early stages of development when many CSCW systems, typically, cannot be tested in their

working environment. It is not easy for designers to determine whether a particular display will satisfy the informational requirements which emerge during the initial design of multi-user applications. This paper argues that designers might exploit formal, mathematically based, specification techniques to resolve these problems during the initial design and formative evaluation of multi-user displays.

Some notable attempts have been made to apply formal methods to support interface development. For instance, Sufrin and He have exploited the Z notation to represent interaction with a text-editor [23]. Harrison and Thimbleby have recruited formal methods to specify high-level requirements for many different interactive systems [9]. Paterno and Faconti have shown how graphical interaction objects can be specified in LOTOS [19]. Unfortunately, formal specifications provide little idea of the 'look and feel' of a CSCW system. Psychological and sociological studies show that such factors frequently determine the success or failure of multi-user systems [10, 21]. Logic programming environments can be used to derive prototypes that provide a much better impression of what it would be like to interact with a potential interface. There is a close correspondence between requirements written in first order predicate logic and programs that satisfy those requirements implemented in PROLOG [4]. Such environments provide means of directly deriving partial implementations from formal specifications. The resulting prototypes can be shown to potential users and are amenable to empirical evaluation during the early stages of development.

A number of problems must be addressed before formal methods can be used to support the development of graphical interfaces to CSCW applications. Firstly, it is unclear how designers might represent images which can be distributed across multi-user systems. The introduction of bitmaps and unstructured graphical representations can threaten the tractability of formal specifications. This, in turn, can obscure important properties of CSCW systems. Secondly, PROLOG was developed to support single-user, single-processor applications. This prevents designers from assessing the 'usability' of graphical interfaces for distributed, multi-user computer systems. Lastly, there is no guarantee that PROLOG prototypes will be correctly transformed into a final implementation. This prevents designers from ensuring that the products of a formal analysis are faithfully carried through into the later stages of development.

This paper describes techniques that avoid the limitations of formal approaches to the design of CSCW systems. Section 3 argues that structured representations, based around regions, can be used to describe distributed, multi-user, displays. Section 4 describes how these abstractions can be directly incorporated into partial implementations which have been developed using the Prelog prototyping tool. It is argued that this tool avoids the single-user, single-processor limitations of PROLOG. Section 5 describes how Prelog displays can be incorporated into full implementations of CSCW systems using the C or Ada programming languages. Section 6 identifies areas for further research and presents the conclusions that can be drawn from this work.

2 An Example Application

Human-computer interfaces to a continuous steel casting control system are used to illustrate the following argument. The casting process exhibits many of the problems that frustrate the design of CSCW systems. Groups of operators must monitor computer displays in order to control several application processes. Operators must detect and coordinate their responses to a range of potential errors. The presentation of such systems, therefore, poses a significant challenge [11]. The casting process is illustrated by Figure 2. Steel from a melting shop 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.

3 Distributed Graphics

The images presented to operators of CSCW systems are qualitatively different to those presented by single-user applications. Some displays are shared amongst the members of a group whilst others are not. Designers must be able to represent and reason about these common and private contexts if they are to provide adequate support for group activities and individual tasks.

3.1 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 inlet icon illustrated in Figure 3 could form part of a *coolant_display* for the casting control system. 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 [24]. 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.

3.2 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:

```
DeclareBitmap(coolant_display.bit, 42, 49, coolant_display.bits);
short on_display.bits[] =
/* Some information omitted 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 gradual introduction of graphical details as development progresses. This also 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.

3.3 Logic Abstractions

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 *coolant_display* whilst *user_2* is presented with a *production_display*. Similar clauses might be introduced to represent the images presented to *user_3*, *user_4*, *user_5* etc:

$$display(user_1, coolant_display). \quad (1)$$
$$display(user_2, production_display). \quad (2)$$

The first clause states that *user_1* is presented with the *coolant_display*. The second clause states that *user_2* is presented with the *production_display*.

Abstractions, such as *production_display*, can be decomposed into their component parts. For instance, the *coolant_display* presented to *user_1* might show that coolant tank A, coolant tank B, water jet B and the pump inlet are all functioning correctly but that there is an error with water jet A. This image is illustrated in Figure 4. The structure of the *user_1*'s *coolant_display* is represented by the following clauses:

$$part(user_1, coolant_display, jet_A_error). \quad (3)$$
$$part(user_1, coolant_display, inlet_ok). \quad (4)$$
$$part(user_1, coolant_display, tank_A_ok). \quad (5)$$
$$part(user_1, coolant_display, jet_B_ok). \quad (6)$$
$$part(user_1, coolant_display, tank_B_ok). \quad (7)$$

The first clause states that the *jet_A_error* icon is part of the *coolant_display* presented to *user_1*. The second clause states that the *inlet_ok* icon is part of the *coolant_display* presented to *user_1*. The third clause states that the *tank_A_ok* icon is part of the *coolant_display* presented to *user_1*. The fourth clause states that the *jet_B_ok* icon is part of the *coolant_display* presented to *user_1*. The fifth clause states that the *tank_B_ok* icon is part of the *coolant_display* presented to *user_1*.

Figure 5 illustrates the *production_display* presented to *user_2*. The structure of this image can be represented in a similar fashion:

$$part(user_2, production_display, jet_A_error). \quad (8)$$
$$part(user_2, production_display, jet_B_ok). \quad (9)$$
$$part(user_2, production_display, cutter_ok). \quad (10)$$
$$part(user_2, production_display, ladle_ok). \quad (11)$$

The first clause states that the *jet_A_error* icon is part of the *production_display* presented to *user_2*. The second clause states that the *jet_B_ok* icon is part of the *production_display* presented to *user_2*. The third clause states that the *cutter_ok* icon is part of the *production_display* presented to *user_2*. The fourth clause states that the *ladle_ok* icon is part of the *production_display* presented to *user_2*.

Designers can use logic clauses to identify those images, such as *jet_A_error* and *jet_B_ok*, 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 jet A is closed if *user_1* and *user_2* are presented with an error for this component and both provide input to close the jet. Section 4 will argue

that prototype implementations can be used to determine whether the image of the water jet will be sufficient to elicit the required responses from $user_1$ and $user_2$. For now it is sufficient to realise that the display requirement is intended to ensure that both operators are presented with sufficient contextual information in order for them to coordinate their response:

$$\begin{aligned} voting_close_jet_A :- \\ & display(user_1, coolant_display), \\ & display(user_2, production_display), \\ & part(user_1, coolant_display, jet_A_error), \\ & part(user_2, production_display, jet_A_error), \\ & input(user_1, 'close_jet_A'), \\ & input(user_2, 'close_jet_A'). \end{aligned} \tag{12}$$

This states that a voting system is used to close jet_A if $user_1$ is presented with the $coolant_display$ and $user_2$ is presented with the $production_display$ and jet_A_error is part of the coolant and production displays and both $user_1$ and $user_2$ provide input to close jet_A.

Such clauses support further stages in the development of CSCW systems. For instance, it has not been specified that the $production_display$ presents information about the pump inlet. It would not, therefore, be appropriate to expect $user_2$ to resolve problems with that component without access to additional data. Contention might occur if they did attempt to operate the inlet. For example, $user_2$ might *close* it whilst $user_1$ tried to *open* it. This would lead to deadlock if neither could progress until the other abandoned their attempt. The display abstractions introduced in the previous paragraphs might be used to specify solutions for such problems. Designers might require that a lock is imposed to resolve contention if $user_2$ is not presented with a $inlet_error$:

$$\begin{aligned} lock_out_inlet_contention :- \\ & display(user_2, production_display), \\ & not(part(user_2, production_display, inlet_error)), \\ & not(input(user_2, 'close_inlet')). \end{aligned} \tag{13}$$

This states that a lock is imposed to prevent contention over the inlet if $user_2$ is presented with the $production_display$ and the $inlet_error$ icon is not part of that display and $user_2$ does not provide input to close the inlet.

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 $production_display$, into the primitive images which are presented to users. One means of doing this is to describe images in terms of lines:

$$line(user_1, inlet_ok, 0.1, 0.2, 0.6, 0.2). \tag{14}$$

This states that the image of the $inlet_ok$ 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 inlet 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 6 illustrates how the image of the *coolant_display* can be described in terms of a number of regions: a background region; a text region and an inlet 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 behaviour, such as whether or not it is selectable. For instance, the *inlet* could be presented to *user_1* as a region with a blank background and dimensions that occupy one twentieth of the screen:

$$dimension(user_1, inlet_ok, 0.05, 0.05). \quad (15)$$
$$pattern(user_1, inlet_ok, blank). \quad (16)$$

The first clause states that the image of the *inlet_ok* icon presented to *user_1* has dimension that occupy one twentieth of the screen. The second clause states that the image of the *inlet_ok* 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 inlet whilst performing other duties. Under such circumstances, the inlet might be introduced into the *production_display*. The dimensions of the inlet icon could also be reduced in order to free display resources for the presentation of *user_2*'s primary activities:

$$part(user_2, production_display, inlet_ok). \quad (17)$$
$$dimension(user_2, inlet_ok, 0.02, 0.02). \quad (18)$$

The first clause states that the *inlet_ok* icon is part of the *production_display* presented to *user_2*. The second clause states that the image of the *inlet_ok* icon presented to *user_2* has dimension that occupy one fiftieth of the screen.

This illustrates how logic abstractions can be used to represent and reason about CSCW systems which support semi-independent views of application processes [6].

3.4 Introducing Input Information

This paper focuses upon the presentation of multi-user applications. It is important to realise, however, that the choice of input media also 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 [7]. 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 trade-offs that exist between tracker-balls, mice, joysticks and keyboards, the designers of CSCW systems must be able to represent a variety of input devices.

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[1]:

```
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
```

[1] Apple Macintosh is a registered trademark of Apple Computer, Inc. [20]

Burton, Cook, Gikas, Rowson and Sommerville show how designers might formalise similar code in order to specify single-user graphical interfaces built from the Apple Macintosh Toolbox [2]. 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.

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, designers might specify that water jet A is closed if *user_1* and *user_2* are presented with a *jet_A_error* and both operators use a mouse to select this image:

$$
\begin{aligned}
&event_close_jet_A :- \\
&\quad display(user_1, coolant_display), \\
&\quad display(user_2, production_display), \\
&\quad part(user_1, coolant_display, jet_A_error), \\
&\quad part(user_2, production_display, jet_A_error), \\
&\quad input(user_1, jet_A_error, on_select), \\
&\quad input(user_2, jet_A_error, on_select). \qquad (19)
\end{aligned}
$$

This states that input events are used to close *jet_A* if *user_1* is presented with the *coolant_display* and *user_2* is presented with the *production_display* and the *jet_A_error* icon is part of the coolant and production displays and both *user_1* and *user_2* issue select events for that icon.

This clause also specifies interfaces in which the users 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.

4 Prelog Prototypes

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.

4.1 Presenting Graphical Structures

Environments such as PROLOG, typically, only provide limited input and output facilities. The Prelog prototyping tool avoids this limitation by linking PROLOG and Presenter [25]. 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 7 shows a display that was generated using Prelog. In order to produce such an image, Prelog constructs a *part* hierarchy using clauses such as (7). The region properties, attributes and behaviours of each part, represented by clauses such as (16), 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 PROLOG and Presenter is to provide the impression of a graphical output channel.

4.2 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. 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 [25]. 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 inlet 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.

4.3 Handling Distribution And Concurrency

PROLOG was intended to run on single-user, single-processor implementations. Some attempts have been made to develop distributed, interactive applications using logic programming environments. Cleary's graphics system for PROLOG exploits distributed processing to provide co-processor support for geometric transformations [5]. The image of the system is not distributed and interaction is focussed through a single monitor. In contrast, our implementation of Prelog extends PROLOG to support interaction between a number of users communicating over local and wide area networks. Figure 8 illustrates the Prelog architecture. It is based around the client and server model. The Prelog server is assumed to be

the logic specification running under PROLOG. Clients are asynchronous processes running on the users' workstations. This architecture has advantages for user testing; the designer can monitor the execution of a prototype on a remote machine. Client processes store and dispatch operator input to the server which distributes shared information to other clients, as required by the logic specification. Client processes are also responsible for handling the presentation of information sent by Prelog's server. Prelog has been implemented on a network of Sun Microsystem's 3/50s and 3/60s, communication is via sockets [17]. Briefly, sockets are the primary form of inter-process communication in SunOS[2] and 4.2BSD versions of the UNIX operating system [3]. They can be thought of as the end-points of two way communications paths. Prelog's client processes can be created on the same or on other machines. The server and its clients must, however, all be launched in the same 4.3BSD communications domain. These are the UNIX domain (local internal UNIX protocol), the Internet domain (using DARPA standard protocols) and the NS domain (using Xerox standard protocols).

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. $graphics_read(From_client, Messa$ is evaluated as true if $Message$ unifies with input sent from a client process. For example, $event_close_jet_A$ (19) 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:

$$
\begin{aligned}
&close_dialogue :- \\
&\quad graphics_write(user_1,'display(coolant_display)'), \\
&\quad graphics_write(user_2,'display(production_display)'), \\
&\quad graphics_write(user_1,'part(coolant_display, jet_A_error)'), \\
&\quad graphics_write(user_2,'part(production_display, jet_A_error)'), \\
&\quad graphics_read(user_1,'input(jet_A_error, on_select)'), \\
&\quad graphics_read(user_2,'input(jet_A_error, on_select)'). \qquad (20)
\end{aligned}
$$

This states that there is a dialogue to close jet_A if a message is written to $user_1$'s client to ensure that they are presented with the $coolant_display$ and a message is written to $user_2$'s client to ensure that they are presented with the $production_display$ and messages are sent to ensure that the jet_A_error icon is part of the coolant and production displays and select events for that icon are read from the $user_1$ and $user_2$ clients.

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 parameterised by their intended destination; $user_1_window_1$.

Extending a logic programming environment to handle interaction between multiple concurrent users raises many implementation problems. Not the least of these is that PROLOG proofs can be slow. This is a critical issue for the evaluation of interactive systems. Users are likely to criticise the deficiencies of the prototyping environment rather than the deficiencies of a potential interface [15]. Our implementation has speeded response times by porting the Prelog server to run on Sun Microsystem's SPARCstations. The flexibility of the socket communications protocols means that hardware support can be obtained through minimal changes to our implementation. A further limitation with our approach is that designers may not be able to reproduce prototype displays within full implementations of CSCW systems. Prelog has been enhanced to avoid this problem.

[2] SunOS is a registered trademark of Sun Microsystems, Incorporated.
[3] UNIX is a registered trademark of A.T. & T. Bell Laboratories.

5 Implementation

Trivial changes were required in order to introduce Prelog displays into full implementations of CSCW systems. The clients are asynchronous independent processes and are easily separated from the Prelog server. Designers can then replace executable logic specifications with applications written in either C or Ada. In terms of Figure 8, the rectangle labelled PROLOG should now be labelled with either Ada or C. This transformation can be illustrated by the following C code which implements the *close_dialogue* (20) clause:

```
void close_dialogue_c()
/* C refinement of close_dialogue */
{
   graphics_write(user_1, "display(coolant_display)");
   graphics_write(user_2, "display(production_display)");
   graphics_write(user_1, "part(coolant_display, jet_A_error)");
   graphics_write(user_2, "part(production_display, jet_A_error)");
   graphics_read(user_1, "input(jet_A_error, on_select)");
   graphics_read(user_2, "input(jet_A_error, on_select)");
}
```

The following Ada procedure also implements the previous dialogue:

```
with PRESENTER; -- package for communication with Presenter socket
procedure CLOSE_DIALOGUE_A is
-- Ada refinement of close_dialogue
begin
   PRESENTER.GRAPHICS_WRITE(user_1, "display(coolant_display)");
   PRESENTER.GRAPHICS_WRITE(user_2, "display(production_display)");
   PRESENTER.GRAPHICS_WRITE(user_1, "part(coolant_display, jet_A_error)");
   PRESENTER.GRAPHICS_WRITE(user_2, "part(production_display, jet_A_error)");
   PRESENTER.GRAPHICS_READ(user_1, "input(jet_A_error, on_select)");
   PRESENTER.GRAPHICS_READ(user_2, "input(jet_A_error, on_select)");
end SELECT_OPEN_PUMP_INLET;
```

It is important to note that the transformation between a logic specification and a procedural implementation is seldom as straightforward as that shown in the previous example [16]. The architecture of our system does, however, ease this transformation by ensuring that partial and full implementations share common abstractions for the representation of multi-user displays.

6 Conclusion And Further Work

This paper has argued that logic abstractions can be used to represent and reason about presentation techniques for CSCW systems. Structured graphics and region primitives provide means of constructing displays whose component images may or may not be shared between a number of users. Generic events can be associated with regions in order to represent operator input from a number of different devices. Unfortunately, formal specifications provide non-formalists with little idea of what it would be like to interact with a CSCW system. Prototypes provide a much better impression of the 'look and feel' of potential implementations. They are amenable to empirical evaluation and can be shown to operators during the early stages of development when it can be difficult to analyse a design within its working context. We have implemented the Prelog prototyping tool to provide a bridge between logic abstractions and partial implementations. This tool uses socket inter-process communication facilities to link a Prolog

server with independent Presenter processes. A limitation of many prototyping tools is that there is no easy way to refine partial implementations into full systems. Prelog has been designed to reduce this problem by re-coupling the client processes to imperative programming languages. This minimises the transformation between logic specifications of multi-user displays and the statements of C and Ada.

This paper has adopted a systems engineering perspective upon the design of CSCW systems. Brevity has prevented a detailed consideration of the ways in which designers can elicit initial requirements for CSCW systems. This is a research issue in its own right. The TMPI project is currently investigating ways in which CSCW requirements differ from those of single-user interfaces [18]. Similarly, other research projects are investigating ways of linking task analysis and formal specification techniques. The Adept project, funded by the U.K. Department of Trade and Industry, is an example of this. Future research intends to examine ways in which the products of these initiatives might support the approach described in this paper.

There is no notion of sequence in first order logic [13]. Current research is exploiting temporal logic as a means of explicitly representing such information during the design of CSCW systems [12, 14]. This formalism 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'). These logic operators provide means of representing safety requirements, such as *user_1* and *user_2* must always agree before closing the plant down. They can also represent synchronisation requirements, such as *user_1*'s command to close the system down will not take effect until *user_2* has issued input to confirm their request. Prelog already incorporates a temporal logic interpreter. Future work intends to extend this tool to support the specification and implementation of real-time requirements.

Video output is increasingly important in the presentation of CSCW systems [1]. Little work has been done to identify design notations which might be used to represent such output. Logic has, however, been used to support image recognition [3]. The device primitives communicated from various types of cameras can be translated into logic abstractions. Further research intends to investigate whether these techniques can be used in order to represent the video images that are presented to the users of CSCW systems.

References

[1] J.L. Alty. Multimedia - what is it and how do we exploit it? In D. Diaper and N. Hammond, editors, *People And Computers VI: Proceedings Of HCI'91*, pages 31–44. Cambridge University Press, Cambridge, United Kingdom, 1991.

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

[3] R.T. Chin and C.R. Dyer. Model-based recognition in robot vision. *ACM Computing Surveys*, 18(1):67–108, 1986.

[4] K. L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, February 1980.

[5] J.G. Cleary. A distributed graphics system implemented in PROLOG. Technical Report 84 173 31, Department of Computer Science, University Of Calgary, Calgary, Canada, 1984.

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

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

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

[9] M.D. Harrison and H.W. Thimbleby, editors. *Formal Methods In Human Computer Interaction.* Cambridge University Press, Cambridge, United Kingdom, 1989.

[10] C. Heath and P. Luff. Collaborative activity and technological design: Task coordination in London Underground control rooms. Technical report, Rank Xerox EuroParc, Cambridge, United Kingdom, 1991.

[11] J.-M. Hoc. 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, *Human-Computer Interaction—INTERACT'87*, pages 257 – 262. Elsevier Science Publications, North Holland, Netherlands, 1987.

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

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

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

[15] P. Johnson. *Human Computer Interaction: Psychology, Task Analysis and Software Engineering.* McGraw-Hill, London, United Kingdom, 1992.

[16] R. Kowalski. Algorithm = logic + control. *Communications Of The ACM*, 22(7):424 – 436, 1979.

[17] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design And Implementation Of The 4.3BSD UNIX Operating System.* Addison Wesley, Reading, United States of America, 1990.

[18] D. Murray and B. Hewitt. Capturing interactions: Requirements for CSCW. In D. Rosenberg and C. Hutchison, editors, *Design Issues In CSCW.* Springer Verlag, Berlin, Germany, 1993 (forthcoming).

[19] F. Paterno and G. Faconti. On the use of LOTOS to describe graphical interaction. In A. Monk, D. Diaper, and M.D. Harrison, editors, *People And Computers VII*, pages 155–173. Cambridge University Press, Cambridge, United Kingdom, 1992.

[20] C. Rose. *Inside The Apple Macintosh*, volume I. Addison Wesley, Wokingham, United Kingdom, 1986.

[21] I. Sommerville, T. Rodden, P. Sawyer, and R. Bentley. Sociologists can be surprisingly useful in interactive systems design. In A. Monk, D. Diaper, and M. Harrison, editors, *People And Computers VII: Proceedings Of HCI'92*, pages 341–353. Cambridge University Press, Cambridge, United Kingdom, 1992.

[22] M. Stefik, G. Foster, D.G. Bobrow, K. Hahn, S. Lanning, and L. Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications Of The ACM*, 30(1):32–47, 1987.

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

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

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

[26] R.C. Williges, B.H. Williges, and J. Elkerton. Software interface design. In G. Salvendy, editor, *The Handbook Of Human Factors*, pages 1416–1449. Wiley, New York, United States of America, 1987.

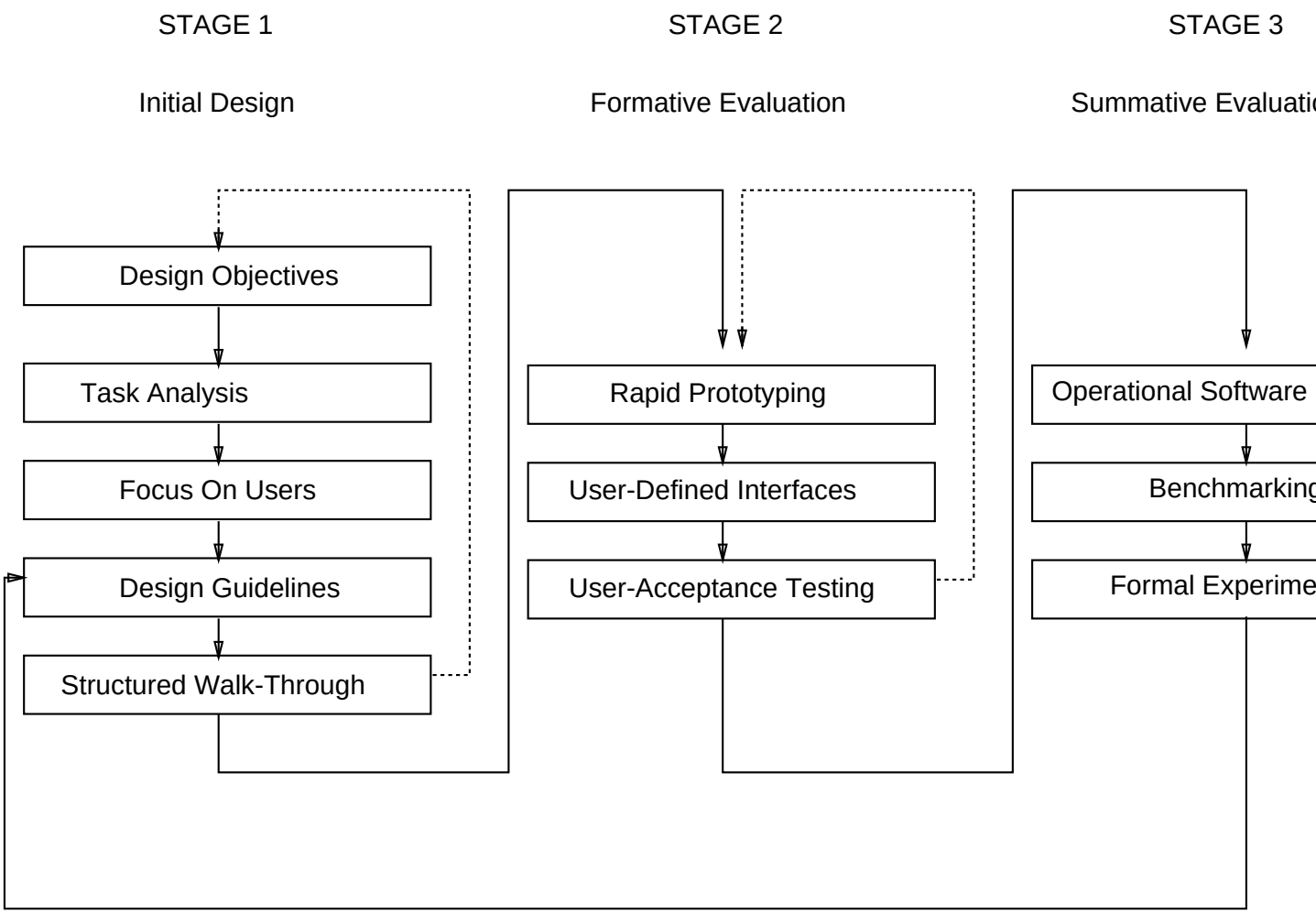

Figure 1: A three stage model of interface development

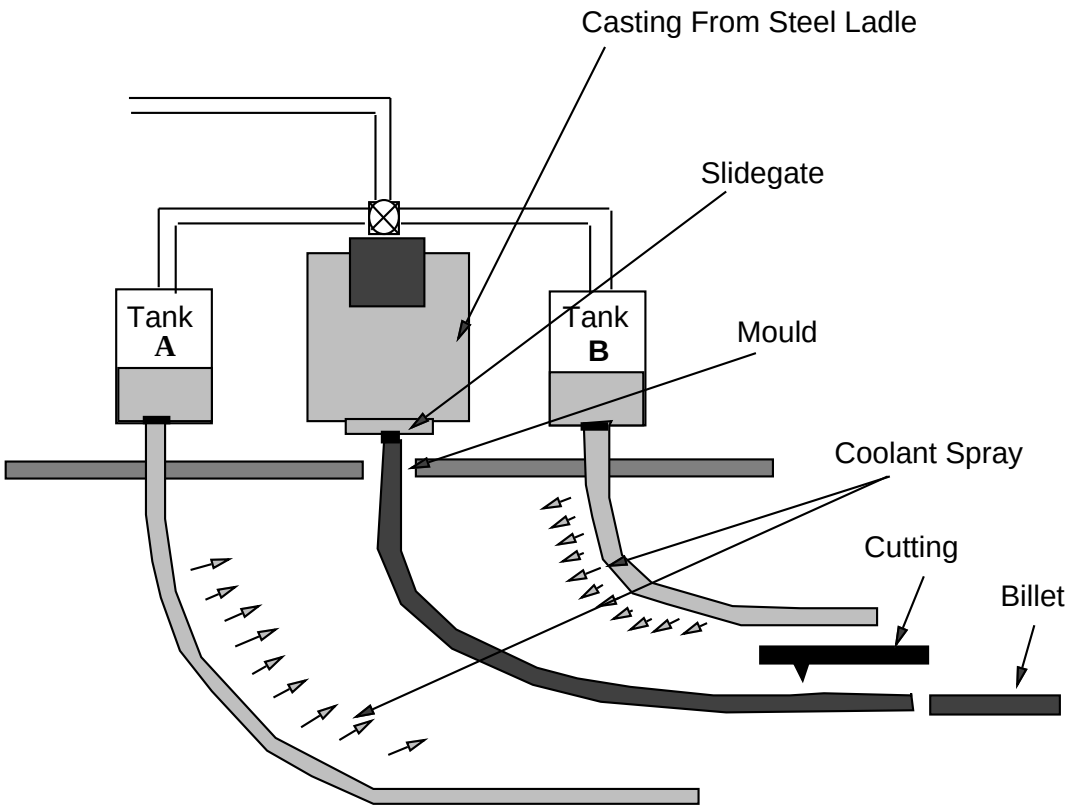

Figure 2: The continuous casting process

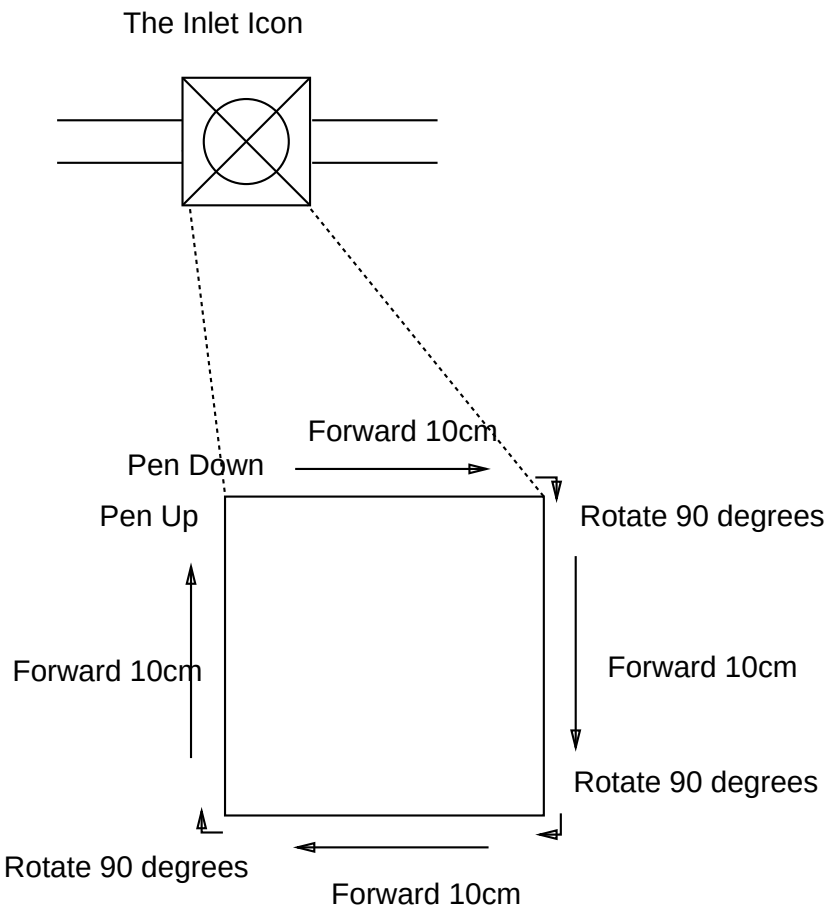

Figure 3: An *inlet_ok* image

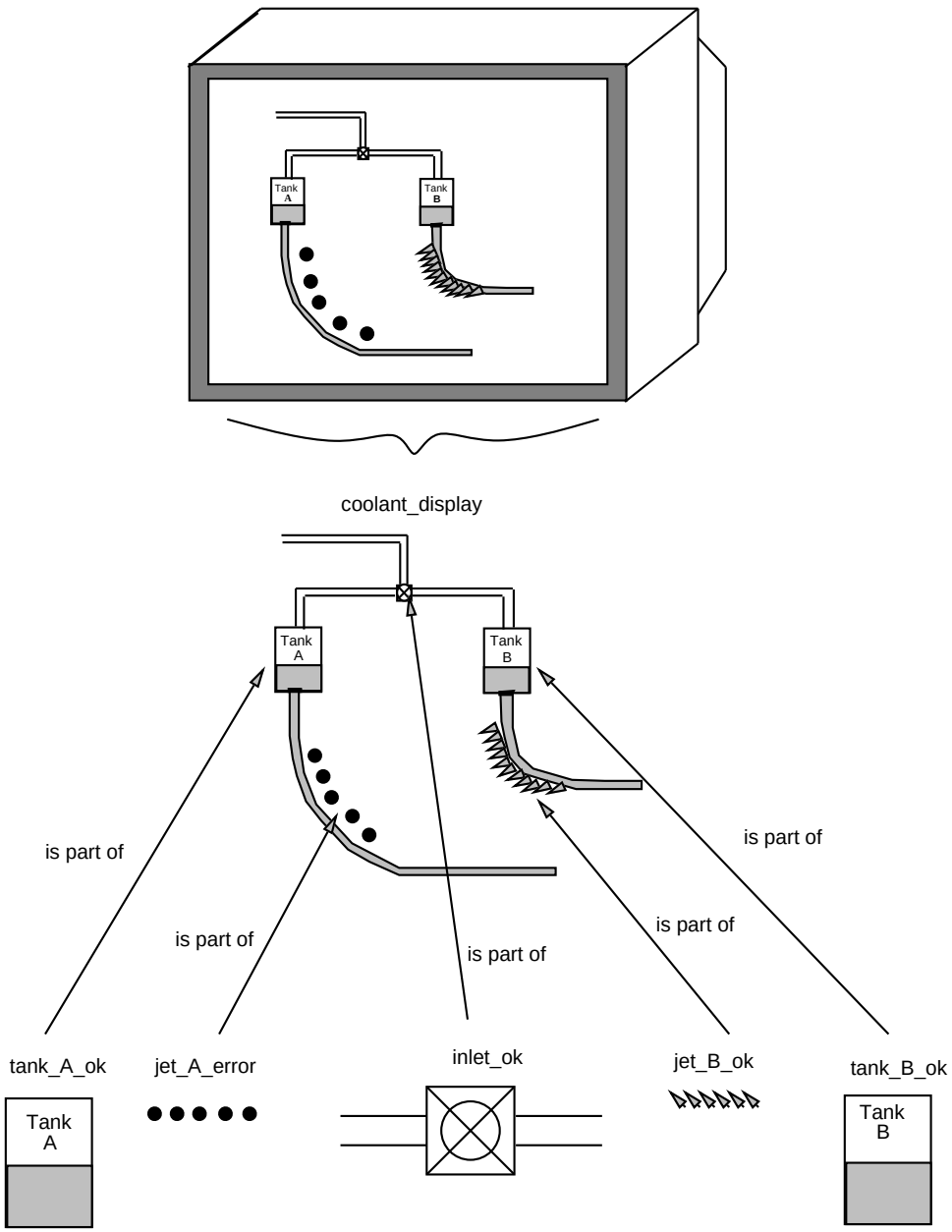

Figure 4: The graphical decomposition of the *coolant_display*

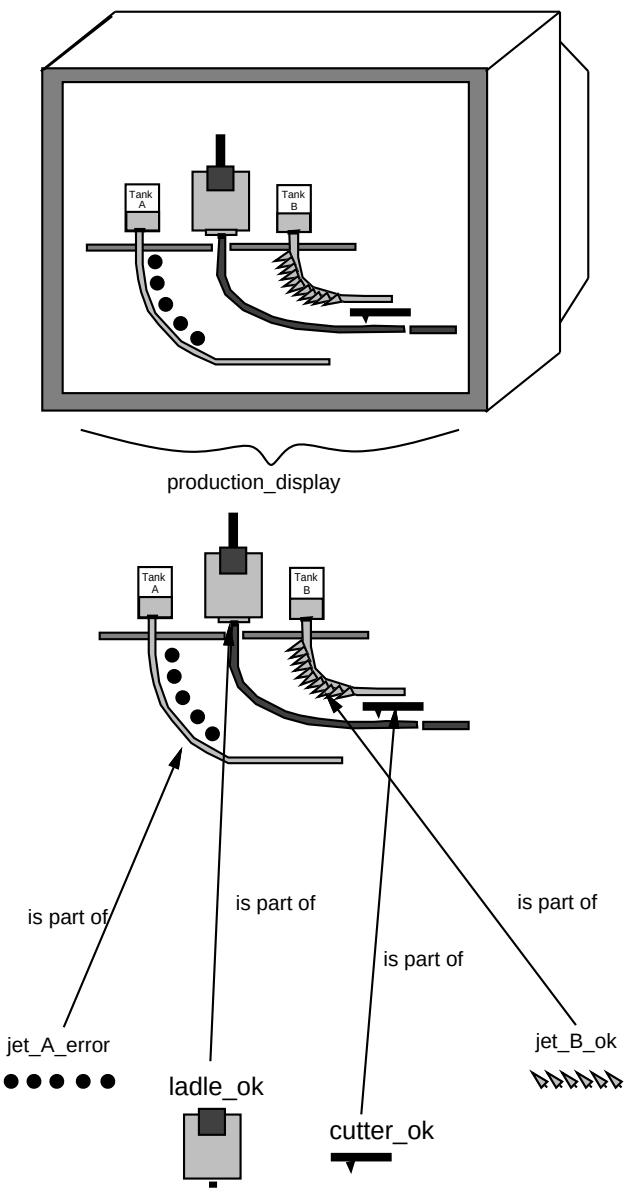

Figure 5: The graphical decomposition of the *production_display*

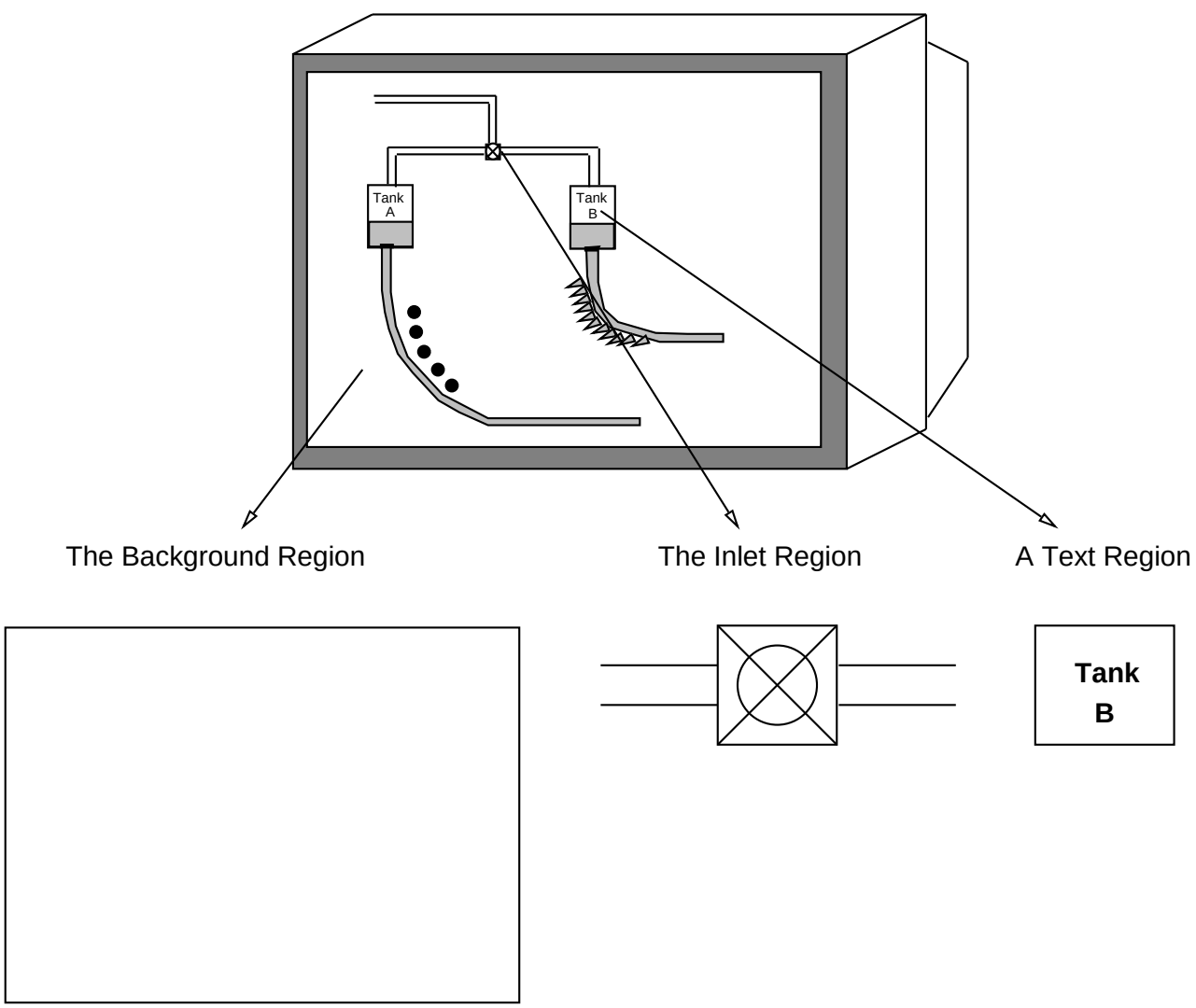

Figure 6: The region decomposition for part of the *coolant_display*

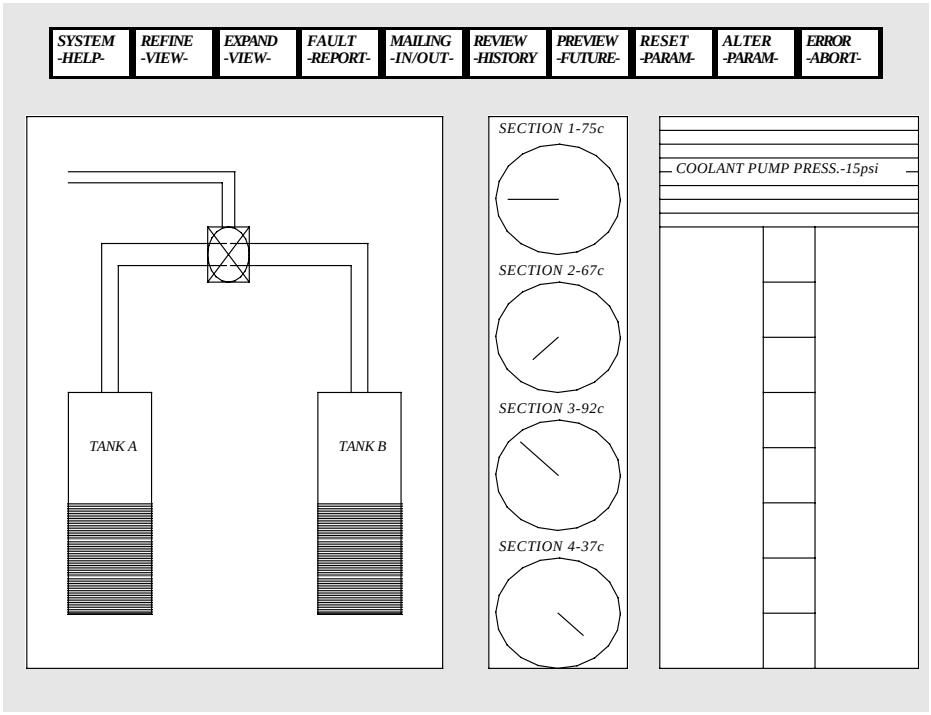

Figure 7: A Prelog prototype

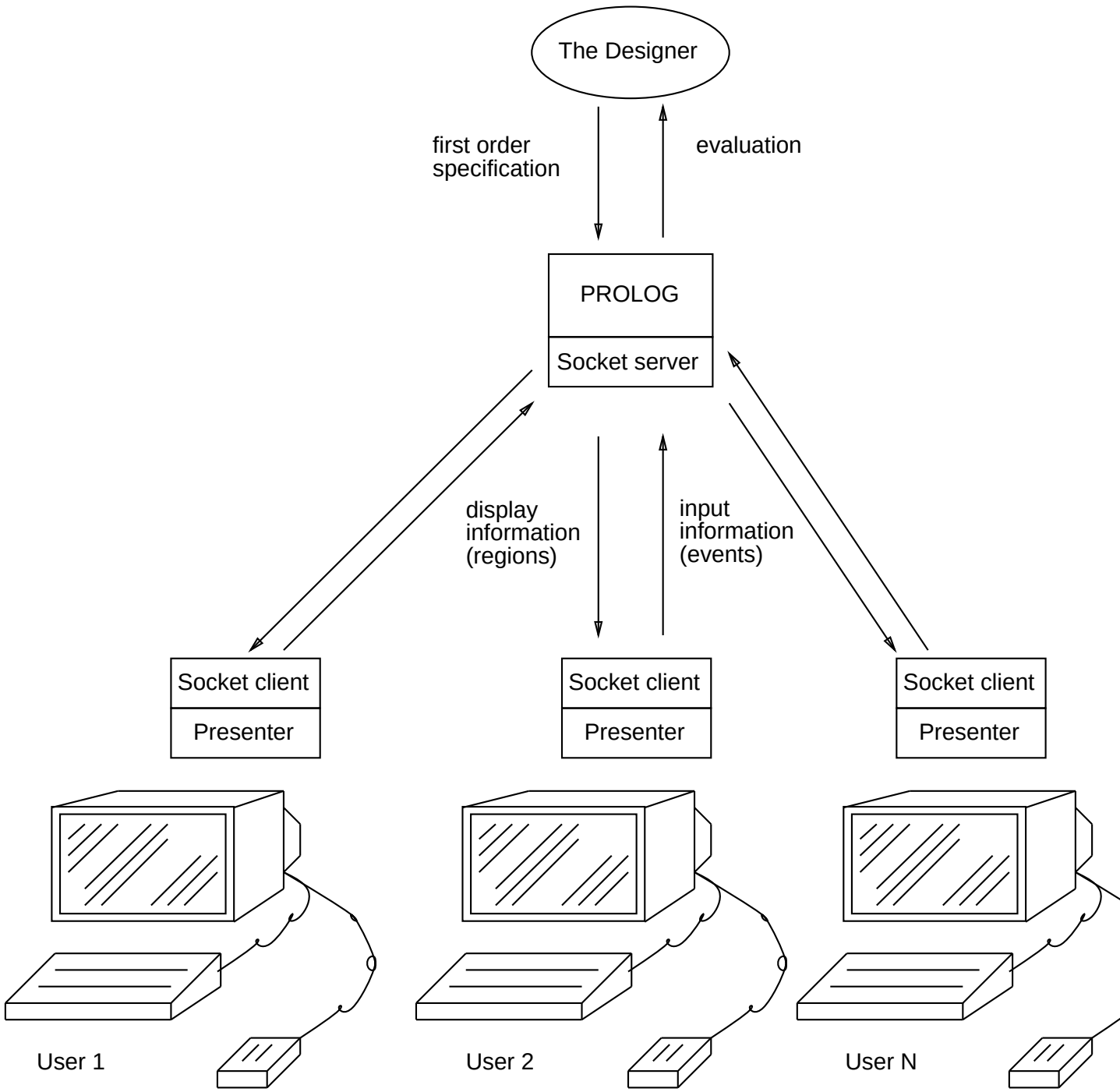

Figure 8: The distributed Prelog architecture