The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation

by

T.C. Nicholas Graham, Herbert Damker[1],
Catherine A. Morton, Eric Telford and Tore Urnes
Technical Report No. CS-96-04
1996

Department of Computer Science
York University
North York, Ontario
Canada M3J 1P3

---

[1]Author's Affiliation: Institut für Informatik und Gesellschaft der Albert-Ludwigs Universität Freiburg, Abteilung Telematik, Friedrichstr. 50, D-79098 Freiburg, Germany, damker@iig.uni-freiburg.de

Abstract

Hartson, Siochi and Hix's User Action Notation allows development teams to specify and evaluate user interface designs prior to their implementation. The UAN is a key feature of user-centered design methodologies, allowing non-programmers to take an active role in the design process. Developers, however, have been slow to adopt the UAN, largely because they do not see it as contributing to implementation. This paper shows how UAN specifications can be methodically transformed into user interface implementations, helping programmers to exploit the information contained in UAN specifications. The paper shows the importance of modern programming language features in supporting this process, in particular, the importance of separating architecture from code, using constraints to separate input from output specifications, and providing high level support for concurrent user interfaces. These features are present in a number of modern research tools, including the Clock language used in the paper. Our method has been applied to the development of over thirty interactive systems, including three well-documented case studies.

Keywords: User Action Notation, User-Centered Design, User Interface Tools, Processes for Groupware Development

Contents

1 Introduction

It is well understood that the design and development of user interfaces is difficult [20]. User interfaces cannot be designed *a priori* and implemented: they must instead be iteratively refined, involving implementation, testing with users, and redesign. Because of the expense of iterative refinement, much effort has been expended in developing techniques of evaluating user interfaces before they are implemented. Additionally, user interface design involves the coordinated participation of people with a variety of backgrounds: in addition to computer scientists, the development group should ideally include domain experts, graphic designers, human-factors specialists and representatives of the ultimate user group. The process of user interface development therefore requires formal techniques for describing and evaluating user interfaces that do not necessarily assume programming knowledge.

Hartson, Siochi and Hix developed the User Action Notation (UAN) [11] as an attempt to bridge the communication gap between programmers and other members of user interface development teams. The UAN permits the specification of user interfaces and provides a basis for evaluating designs. The key idea of the UAN is that it records a *task-oriented* specification of user interfaces, where for every task a user may wish to perform, the UAN is used to record how the task would be carried out. This approach specifies the behaviour of a user interface by showing how people use the system to achieve their goals. At the same time, UAN specifications help evaluate the effectiveness of a user interface design, by showing which tasks are hard (or even impossible) to carry out. The UAN provides a precise specification of the dynamic behaviour of a user interface, providing the eventual programmer with detail to help with implementation.

The UAN would therefore appear to be a highly useful tool for user interface development. Our initial experience with the notation, however, has been that developers do not like it, do not feel that task-oriented specification contributes enough to the development process to be worth the effort, and would not be likely to use it again.[2] Largely, this unhappiness with the UAN stems from a feeling that UAN specifications do not contribute in a direct way to implementation. UAN specifications are in fact very far from implementations, for two reasons: first, moving from UAN to code requires a change in point of view, from the behavioural view of UAN to the implementation view of code. Secondly, the UAN is much higher level than code, and does not apparently help with the programming problems of modern user interfaces, such as handling direct-manipulation input, maintaining consistency among user interface components, and synchronizing the concurrent actions of multi-media streams and multiple users operating over a network.

This paper presents an approach to better integrating UAN into the user interface development process. The key to the method is that UAN specifications are directly used to derive user interface implementations. Our presentation is based on the Clock [6] user interface development language. Clock is a declarative language for the development of multi-user, multi-media applications with high-level level support for concurrency, distribution and consistency maintenance over multiple users. Clock has been implemented, and runs over both local and wide-area networks [8]. Since the level of specification in Clock is so much higher than in traditional languages, the main issue of moving from UAN to Clock is the change in point of view, not the change in level of notation. Based on this, we are able to show how translating from task-oriented specification to user interface implementation can be carried out in a methodical fashion, even when the user interface involves multiple users and concurrency.

The key features of Clock that help in this derivation are present in numerous other user interface

---

[2]Our experience with UAN comes from three substantial case studies performed within our own group [2, 19, 26] and from the experience of over 100 students using the technique to develop over 30 systems in a fourth year University course.
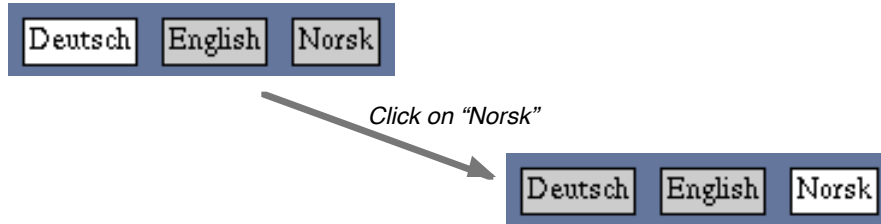
Figure 1: A simple user interface allowing a user to select the language of choice. Clicking on one of the languages changes the selection.

tools and languages. The features we found to be important are: the architecture (or high-level design) of a user interface should be specified separately from the actual user interface code (as in the PAC model [1], the InterViews toolkit [17], the RendezVous language [14] and the CHIRON-I toolkit [25]); input and output should be handled separately through the use of constraints or other forms of implicit invocation (as in the MVC model [16], the Garnet toolkit [21] and RendezVous); and high level support should be provided for concurrency and concurrency control. We therefore believe that this method should be applicable using target languages other than Clock.

In summary, the primary contributions of this paper are:

- We show how UAN specifications can be transformed in a methodical manner into user interface implementations, helping programmers produce better implementations faster. This derivation of implementations from task-oriented specifications helps demonstrate to programmers the utility of task-oriented specification, and encourages them to participate in a user-centered design process.

- Through the process of deriving implementations, we demonstrate what properties are desirable in a language for implementing user interfaces. Researchers in user interface tools have been converging on a similar set of desirable properties. Our work therefore provides a new validation of the direction of user interface tools research.

- Based on our experience with using the UAN in the development of over thirty interactive systems over the last five years (including three well-documented case studies [2, 19, 26]), we discuss the strengths and weaknesses of the notation.

The paper is organized as follows. Section 2 presents an overview of the Clock methodology for deriving user interface implementations from specifications, introducing the UAN and Clock notations. Section 3 describes the User Action Notation in detail, showing how it can be used to specify the behaviour of a simple groupware application, and analyzes the strengths and weaknesses of the notation. Section 4 then shows how the UAN notation can be used to methodically derive implementations in high level languages such as Clock. Section 5 then shows how the difficult problems of concurrency are handled by the method.

2   Task-Oriented and Constructional Specifications

This section introduces the Clock methodology for deriving implementations from task-oriented specifications. This overview briefly introduces the UAN notation for task-oriented specification
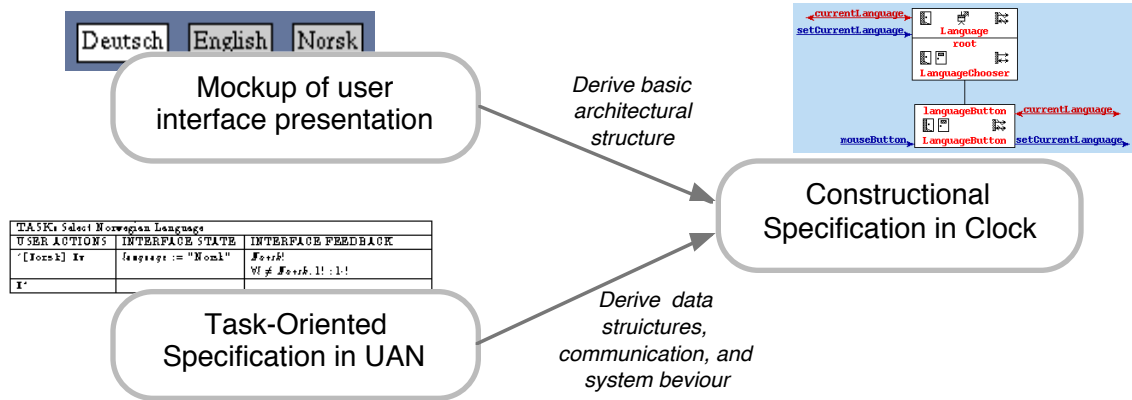
Figure 2: The process of deriving implementations from design documents.

TASK: Select Norwegian Language		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
~[Norsk] Mv	$language :=$ "Norsk"	$Norsk$! $\forall b \neq Norsk.\ b!\ :\ b$-!
M^		

Figure 3: UAN specification of the task of setting the language to Norwegian.

and the Clock language for user interface implementation, and shows how information in UAN specifications can be used to derive Clock programs. Section 3 will describe the User Action Notation in greater detail, while section 4 describes the derivation process in more detail.

This overview shows the derivation of a simple interaction technique, shown in figure 1, that allows users to select their language of choice among German, English and Norwegian. The currently selected language is always shown with a white button. To change language, a user simply clicks on the desired button.

In the Clock methodology, information contained in user interface design documents is used to derive implementations. High level features of Clock (also found in other modern user interface development tools) simplify this derivation. Since the methodology allows design documents to be of direct use in implementation, programmers are more likely to buy in to a user-centered design process.

Figure 2 shows the process of deriving implementations of interactive systems from designs. Mock ups of the user interface (such as those of figure 1) are used to specify the graphical presentation of the user interface. A task-oriented specification in the UAN records how the user performs his/her tasks with the system, thereby providing a dialogue-level specification. Together, these capture a complete specification of the system which can be used to derive an implementation in Clock. We shall first present how UAN is used to develop a task-oriented specification, and then show how this specification leads to a Clock implementation.
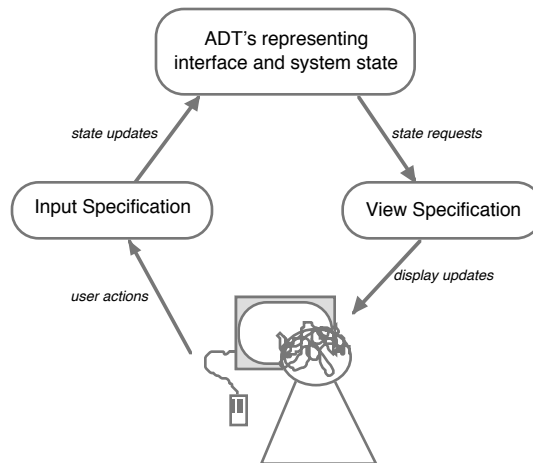
Figure 4: The model underlying Clock programs.

2.1   User Action Notation

Hartson, Siochi and Hix [11] pioneered the concept of *task-oriented* specifications of user interfaces using their *User Action Notation*, or *UAN*. A task-oriented specification shows how a system is used to perform tasks of interest to its users. As such, a task-oriented specification forms a precise specification of the user interface of a system, but from the point of view of a user.

Figure 3 shows a UAN specification recording what actions a user must perform to change the current language to Norwegian. The UAN chart breaks the specification into *user actions*, specifying what the user must do to perform the task, *interface state*, showing how the user's actions affect the internal state of the user interface, and *interface feedback*, specifying the visible effects of these actions.

The specification shows that to select the Norwegian language, the user first moves the mouse over the *"Norsk"* button, as indicated by the special symbol "`~[Norsk]`". The user then depresses the mouse button ("`Mv`"). When the button is depressed, the current language of the user interface is changed to Norwegian. This is recorded using the pseudo-code notation of "*language* := "Norsk" ". The *interface feedback* column records that the *Norsk* button is highlighted (*"Norsk !"*), and that whatever other button was highlighted is dehighlighted (*"b-!"*). The user then releases the mouse button ("`M^`"), which has no further effect on the user interface.

As will be seen in section 3, the UAN notation allows the expression of complex tasks and user actions, including the concurrent interaction of multiple users.

A complete UAN specification shows how each task of interest to the user is carried out with the system. UAN is therefore useful in evaluating a user interface, since a complete specification shows whether all tasks of interest to the user can be accomplished in a reasonable way. UAN also helps evaluate the consistency of a user interface, showing whether similar tasks are carried out in similar ways.
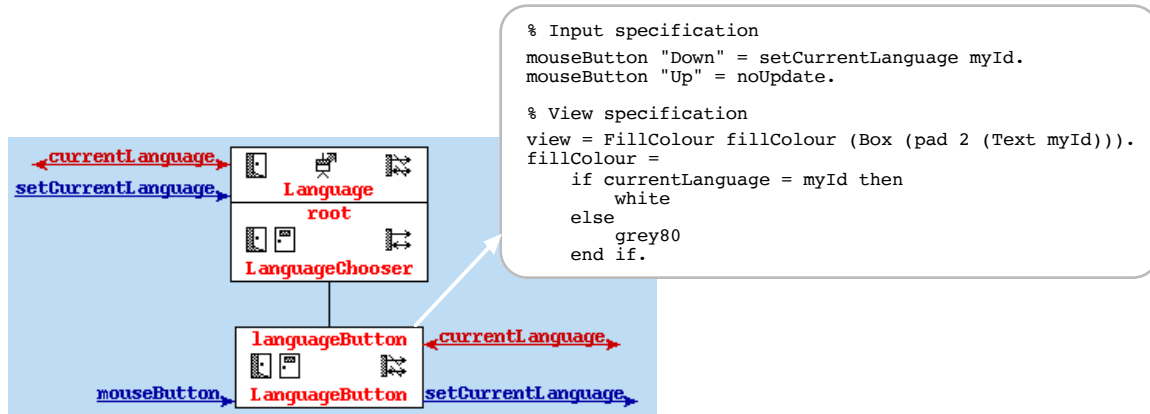
```
% Input specification
mouseButton "Down" = setCurrentLanguage myId.
mouseButton "Up" = noUpdate.

% View specification
view = FillColour fillColour (Box (pad 2 (Text myId))).
fillColour =
    if currentLanguage = myId then
        white
    else
        grey80
    end if.
```

Figure 5: Clock code implementing UAN specification of figure 3.

2.2 Clock

Clock programs are specificational. As opposed to the task-oriented specification of UAN, however, Clock specifications are *constructional*, or from the point of view of the system. In Clock, for each component of the user interface, a specification is given stating how the component responds to input, and a view specification states how the component appears on the display (figure 4). This program organization completely separates input from output: whenever an input modifies the system state, the runtime system automatically updates the display to maintain its consistency with the view specification. This implicit triggering of displays is derived from the Smalltalk Model-View-Controller paradigm [16], and is found in other constraint-based user interface tools, such as Garnet [21], RendezVous [14] and Amulet [18].

Figure 5 shows how this UAN chart is implemented in Clock. The high-level structure of Clock programs is specified in a visual architecture language using the ClockWorks [7] visual programming environment. Clock architectures consist of a tree of components, arranged according to the compositional structure of the user interface. Here, the user interface consists of three radio buttons, represented by the *LanguageChooser* component. The radio buttons are composed of three buttons, each of which is represented by an instance of the *LanguageButton* component. Therefore, a parent component (such as *LanguageChooser*) may compose any number of instances of its children (here, *LanguageButton*) in creating its own view.

Components communicate via messages, termed *requests* (requests for information), and *updates* (state modification). The arrows on the right side of a component indicate the messages a component uses, while the arrows on the left side indicate the messages a component is capable of handling. Therefore, each instance of the *LanguageButton* component can respond to mouse button input directed to it, may request which language is currently selected (via the *currentLanguage* request), and may update the current language (via the *setCurrentLanguage* update.)

Attached to components may be Abstract Data Types (ADT's), which are used to represent state. Here, the *Language* ADT is used to represent which language is currently selected, and implements the *currentLanguage* request and the *setCurrentLanguage* update.

Components in Clock contain specifications of how the component responds to inputs directed to

9

it, and how the component appears on the display. These specifications are encoded in a functional language similar to Haskell [15]. As an example, figure 5 shows the implementation of the *LanguageButton* component. Intuitively, this specification states that whenever a button with text *id* is clicked, then *id* should become the current language. The view of the component is to be a box containing the text *id*; if *id* is the current language, then the button is to be drawn in white, and otherwise in grey.

The following encodes the input specification:

```
mouseButton "Down" = setCurrentLanguage myId.
```

This code simply states that whenever the mouse button is depressed over this component, the current language is to be set to this component's identifier (i.e., either *"Deutsch"*, *"English"* or *"Norsk*, as specified by the request *myId*).

The *view* function is a specification of the component's view:

```
view = FillColour fillColour (Box (pad 2 (Text myId))).
fillColour =
    if currentLanguage = myId then
        white
    else
        grey80
    end if.
```

This function states that the view is a box surrounding the button text, and that the fill colour of the box (either *white* or *grey80*) depends on whether the button represents the current language. The key notion behind this view specification is that the programmer does not need to worry about how the specification is maintained. Whenever the current language changes, the view function is automatically updated so that the button appears in the correct colour. View functions are therefore truly specificational, describing the appearance of the display without giving an algorithm for how display updates are to be implemented.

Transforming UAN specifications to Clock programs is therefore a question of changing one specificational notation to another. UAN specifications are written from the point of view of a user, and are therefore a natural notation for discussing user interface design. Clock programs are written from the point of view of the system, and are therefore natural for discussing implementation. Since both notations are specificational, Clock forms a natural bridge between design and implementation of user interfaces.

2.3   Deriving Implementations from Task-Oriented Specifications

User Action Notation specifications contain information that permits implementations to be derived from specifications in a methodical manner. The Clock methodology for deriving implementations consists of four major activities:

- Use mockups of the user interface presentation to derive a skeleton architecture.

- Use information in the *interface state* column to derive ADT's; use the *user actions*, *interface feedback* and *interface state* columns to position these ADT's in the architecture, and to develop component interfaces.
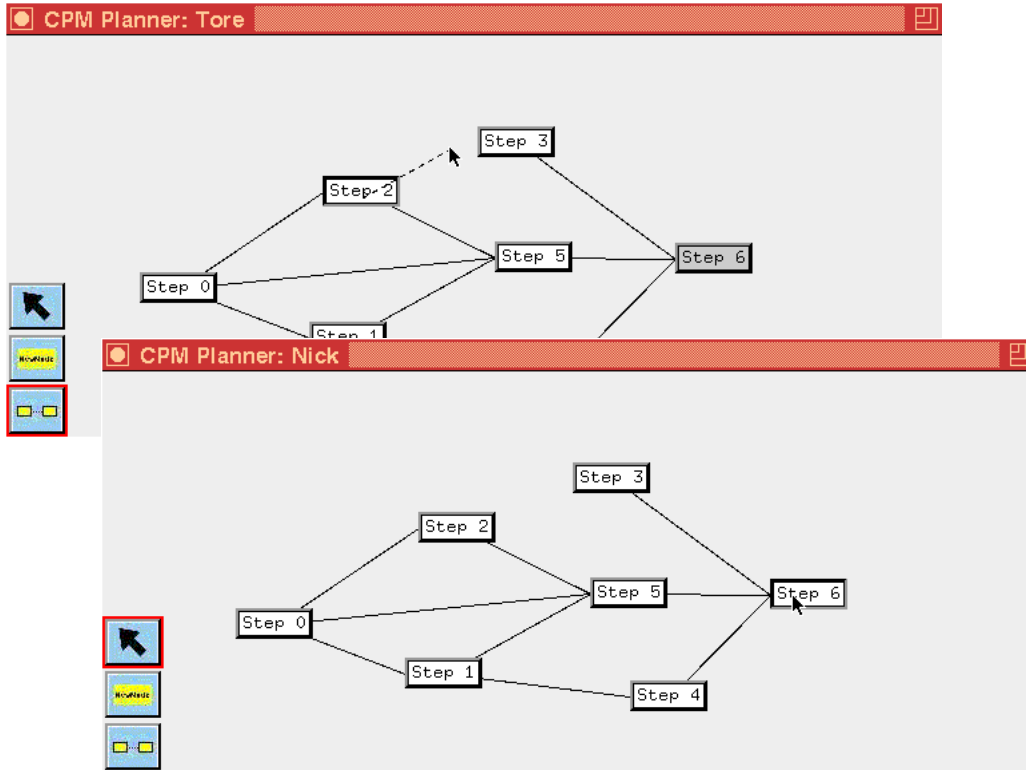
Figure 6: A multiuser project planning application, derived using the Clock methodology. "Tore" is connecting nodes two and three while "Nick" repositions node 6.

- Use decision trees to derive view functions from the *interface feedback* column.

- Use decision trees to derive input functions from the *user actions* column.

These steps are described in detail in section 4, and collected in reference format in appendix A.

This method depends on several features of the Clock language. Clock's separation of architecture from code allows the derivation of the program's skeleton architecture, ADT's and inter-component communication independently of coding details. Clock's separation of input and output allows the derivation of view and input functions to be treated as two separate steps, avoiding the complexities of interaction between input and output. As will be seen in section 5, Clock's implicit concurrency control allows the derivation to avoid complex issues of concurrent behaviour. Since these features are also found in other modern user interface development tools, we believe that the Clock methodology should be adaptable to other target languages.

The next section introduces the User Action Notation in more depth, and presents our experiences with the notation. A simple groupware application is introduced that will be used throughout the paper to illustrate the Clock methodology.
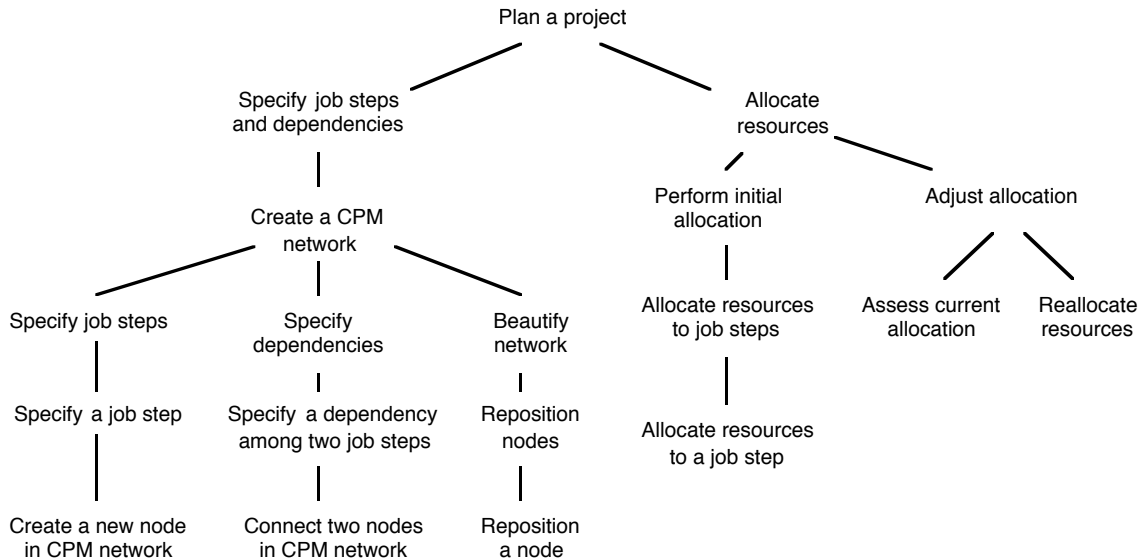
11

Plan a project

Specify job steps
and dependencies

Allocate
resources

Create a CPM
network

Perform initial
allocation

Adjust allocation

Specify job steps

Specify
dependencies

Beautify
network

Allocate resources
to job steps

Assess current
allocation

Reallocate
resources

Specify a job step

Specify a dependency
among two job steps

Reposition
nodes

Allocate resources
to a job step

Create a new node
in CPM network

Connect two nodes
in CPM network

Reposition
a node

Figure 7: Task analysis for planning a project using the CPM method.

3   Task-Oriented Specification in the User Action Notation

This section presents the User Action Notation, and describes how it used to record hierarchical task analysis and to describe and evaluate the behaviour of interactive systems. The section describes why we think the UAN is an important tool in user interface development, but also details the problems we have had with the notation.

As a motivating example, we use the task of planning a project using the critical path method (or *CPM*), as described by Dilworth [3]. Using this method, projects are decomposed into a series of job steps. Dependencies among job steps are recorded by arranging them into a network. Resources are allocated to the job steps, and reallocated based on identification of the critical path through the network.

Figure 6 shows an example CPM planner implemented in Clock. (This program was simplified from the more extensive Clock program for exposition in this paper.) The program permits users to create nodes in a critical path method network. Nodes are assigned numbers sequentially as they are created. Nodes can be linked by drawing a line from the source to the target node, and can be repositioned. Mode buttons select between modes for repositioning, creating and connecting nodes. In a complete CPM planning application, weights are also added to each node, allowing computation of the critical path through the network.

The application is multi-user, meaning that a group of people can collaborate to create a network. Each user sees the same network display. As users move and connect nodes, the other users' displays are updated to reflect the changes. In order to avoid competing actions, a simple locking protocol is used. For example, if a user starts moving a node, no other user can move the same node until the first user has released it. To help participants be aware of the ongoing actions of other people, locked nodes appear with a grey background on the displays of all users other than the lock holder.

The remainder of this section shows how task analysis and task-oriented specification helped in the

TASK: Plan a project
USER ACTIONS
(Identify job steps and dependencies) || (Allocate resources)

TASK: Identify job steps and dependencies
USER ACTIONS
Create a CPM network

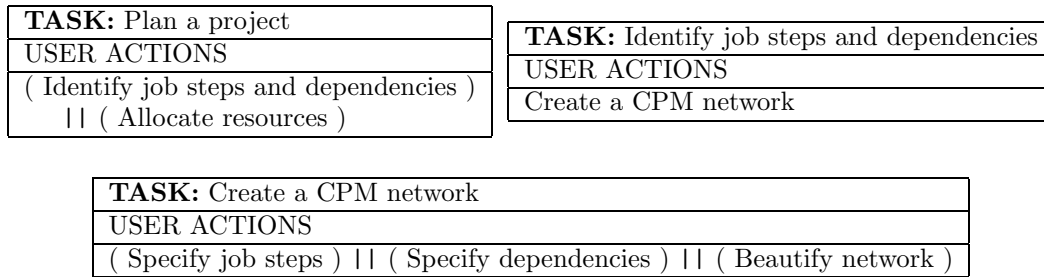| TASK: Create a CPM network |
| --- |
| USER ACTIONS |
| (Specify job steps) \|\| (Specify dependencies) \|\| (Beautify network) |

Figure 8: Top nodes from HTA recoded in UAN

design of this application. Section 4 then shows how the implementation of the application was derived using the Clock methodology.

3.1   Task Analysis

A task-oriented specification of a user interface is derived from a task analysis of the problem domain. A task analysis records all the tasks that a potential user of the system may need to perform. The design of an interactive system is based on supporting these tasks. Figure 7 gives a hierarchical task analysis of the problem of planning a project using the critical path method (*CPM*). (This task analysis was partially derived from Dilworth's description of critical path planning [3].) The root task of planning a project is split into the two subtasks of specifying the steps involved in the project and how they depend on each other, and of allocating resources to each step. (The first of these tasks is supported by the program of figure 6, while the second is not.) The task of specifying job steps and dependencies is solved via the task of creating a CPM network, which in turn consists of the subtasks of specifying the job steps, specifying their dependencies, and beautifying the presentation of the network.

A hierarchical task analysis (e.g., as presented by Preece [24]) describes the tasks people need to perform to do their jobs, not the details of dealing with a particular computer system. The tasks presented in an HTA do not necessarily have to be performed in sequence or even independently. For example, a planner using the CPM method might switch back and forth between adding nodes and addding dependencies. Similarly, when a group of people are planning a project, one person may be specifying dependencies while another beautifies the network. One of the strengths of the UAN notation is that it encodes the information in the task analysis, while specifying precisely how these tasks may be sequenced, interleaved, or performed concurrently.

3.2   Task-Oriented Specifications and UAN

A task-oriented specification of a system in UAN shows how a person uses the system to accomplish his/her goals. A UAN specification starts by encoding the information derived in the hierarchical task analysis. The UAN uses a chart notation to encode each task in the HTA, showing how the task is performed in terms of atomic actions and subtasks. For example, as shown in figure 8, the root task of the HTA is encoded as the *"Plan a project"* chart. The chart states that to plan a project, a user must carry out the actions of identifying job steps and dependencies, and allocating

13

TASK: Reposition a node		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
select mode *"move"*		
~[n] Mv	lock n	$n!$ n locked
(~[x,y]	$nodePos(n) := (x, y)$	display n at (x,y) on all users' displays, moving all connected lines and rubber-band lines.
)* M^	unlock n	n-! n unlocked

Figure 9: UAN for repositioning a node

resources to each job step. The "||" symbol means that these tasks can be carried out concurrently – either by a single user moving back and forth between the tasks, or even by two users working on different parts at the same time.

The *"Plan a project"* task utilizes two subtasks (also appearing in the HTA), each of which is specified through its own chart. For example, the *"Create a CPM Network"* task is carried out by specifying job steps, specifying dependencies among the job steps, and beautifying the network. Once again, these tasks can be carried out in parallel, possibly by different people cooperating on developing the CPM network. In addition to the "||" symbol for specifying that tasks can be performed concurrently, the UAN also provides symbols specifying interleaving ("<=>"), conjunction ("&"), disjunction ("OR"), repetition ("*") and sequencing of tasks. Therefore, the UAN carries more information than a standard HTA, by clearly specifying task dependencies and alternatives between tasks.

As the leaves of the task hierarchy are reached, UAN specifications shift emphasis from describing the task hierarchy itself, to describing how tasks are carried out using the system. For example, figure 9 shows how the task of repositioning a node is carried out in the CPM system. This UAN chart has three columns. The first (*user actions*) shows the actions the user performs to reposition a node. The second (*interface state*) shows how these actions affect the internal state of the user interface. The third (*interface feedback*) shows how the system responds to the user's actions.

The first user action is to select the *move* mode, specified in the *select mode* subtask (not shown here.) The second action is to move over some node n (~[n]), and depress the mouse button (Mv). Clicking down on a node has the effect of locking the node so that nobody else can move it. The visible effects of this action are: the node moves to a sunken relief (n!) to indicate that it has been depressed, and the node appears locked (i.e. with a grey background) on all other users' displays (n locked).

The next line of the UAN chart specifies that as the mouse is moved to each new (x, y) position (~[x,y]), the position of the node is updated, and the node (and all connected lines) are redrawn on each users' display. The movement may occur any number of times (as indicated by the "*".) When the mouse button is released (M^), the node is unlocked, released from its depressed state (n-!), and redrawn as unlocked on all users' displays.

The repositioning task shows that even with something as simple as dragging a node, there is considerable complexity in a multi-user environment. The UAN notation serves to clarify the effects

14

of each stage of the repositioning dialogue.

It is important to note how the UAN simplifies the specification of concurrency. In the task hierarchy of figure 7, *"Reposition a Node"* appears as a subtask of *"Specify job steps and dependencies"*. As was seen in figure 8, this means that multiple users can be concurrently contending to move and connect nodes. UAN charts can be read as guaranteeing atomicity at the level of primitive operations in the user interface. For example, the second line of the *"Reposition a Node"* task consists of the user actions: "`~[n] Mv`". That is, the user moves the mouse over an unlocked node n, and depresses the mouse button. The UAN guarantees that, assuming the node is unlocked at the instant the user depresses the mouse, the subsequent modifications to interface state and interface feedback will be treated as atomic. This guarantees that two users cannot lock the same node at the same time. If the user interface is to provide concurrency control at a higher level than primitive operations, a higher level mechanism (such as the locking applied in this example) must be used. Section 4 provides a more detailed analysis of concurrency in the UAN.

3.2.1 Advantages of UAN

We have found task-oriented specification in UAN provides a wide range of benefits. These include helping to link task analysis and user interface design, helping to implement a user interface design, and in determining how to test a user interface's design and implementation:

Demonstrating task coverage: The purpose of the UAN is to show how each task identified in the hierarchical task analysis can be carried out using the system. This shows whether all tasks are supported in a reasonable and consistent way, and verifies that the system indeed supports the tasks it is meant to support.

Delineation of system's boundaries: Typically, interactive systems do not support all tasks required to carry out a process. A hierarchical task analysis can be used to identify all tasks required by the process. Using the UAN to explicitly show which of these tasks are supported by the system helps in understanding how the system fits within the context of its use.

Feedback on task analysis: Task analysis itself is an iterative process. Performing task-oriented specification presents an opportunity to analyze the correctness and completeness of the task analysis, helping to identify potentially erroneous or unclear features.

Basis for testing: It is very difficult to test that interactive systems have been correctly implemented. While errors in implementing a user interface's presentation are easy to see in a running program, errors in implementation of concurrency or in responding to erroneous use can be hard to find. A UAN specification provides a set of precise test cases, showing realistic sequences of user actions and intended feedback.

Precise specification: UAN provides a useful specification of the operation of the system that can be used by the system's implementers. UAN specifications are more precise than specification by scenarios, mockups or prose. Areas where the UAN is particularly useful include consistency maintenance among different components of a user interface, and concurrency. Both of these areas are particularly of interest to groupware developers, where consistency must be maintained among the user interfaces of different users, and where the concurrent activities of multiple users must be arbitrated.

3.2.2 Disadvantages of UAN

Despite its many advantages, we have found that user interface developers are highly reticent about using UAN in system development. In our four years of experience with variants of the Clock methodology, feedback from designers has consistently indicated that they felt UAN did not help in developing their systems, and that they would not be inclined to use it again. This dislike of UAN comes from a number of sources:

Tedious: UAN specifications are tedious and time consuming to create. Specifications for complex systems can run over dozens of pages. Long specifications become hard to read, requiring indexes to locate subtasks buried within the document. The underlying tree structure of the system can become lost in the many pages of detailed charts. Typesetting UAN is particularly tedious.

Hard to maintain: Even fairly minor changes in the design of a user interface often require substantial changes in the UAN specification. Often it is difficult to isolate all the parts of the UAN where changes are required. Modifying typeset UAN documents is time consuming and tedious. As the design evolves, the UAN is typically left behind, as designers become unwilling to spend the time updating it.

Poor linkage to implementation: Programmers do not clearly see the linkage between UAN specifications and their implementation. The gap between task-oriented specification and implementation is sufficiently large that developers do not see performing the specification as a step towards implementation. Our experience is that implementation is in fact simplified, since programmers have a more clear understanding of the details of the system, but that programmers do not recognize that this simplification has occurred. In short, programmers do not believe that the costs of performing a UAN specification are amortized by reduced development costs.

We believe that the first two points are best addressed through improved tool support for the UAN. To this end, we have developed an experimental hypertext browser for the UAN [26], supporting easy navigation of UAN specifications. Future work involves upgrading this browser to a complete editor.

The methodology presented in this paper aims to address the last point. The Clock methodology links the UAN to implementations by providing programmers with a means of deriving implementations from specifications.

4 Deriving Implementations from UAN Specifications

This section gives a detailed overview of the Clock methodology, showing how user interface implementations can be derived from task-oriented specifications expressed in the UAN. The presentation emphasizes the importance of high-level features of the Clock language, in order to establish that the methodology should be applicable using other languages with similar features.

The goal of the Clock methodology is to convince developers that it is worthwhile to invest time in creating and maintaining UAN specifications. As is shown in this section, programmers benefit from specifying complex user interface behaviour in the UAN prior to implementation. The UAN notation provides the context in which user actions may occur. This simplifies the description of user actions that may have different meanings in different contexts, and simplifies the description of tasks that may be interleaved or carried out concurrently.
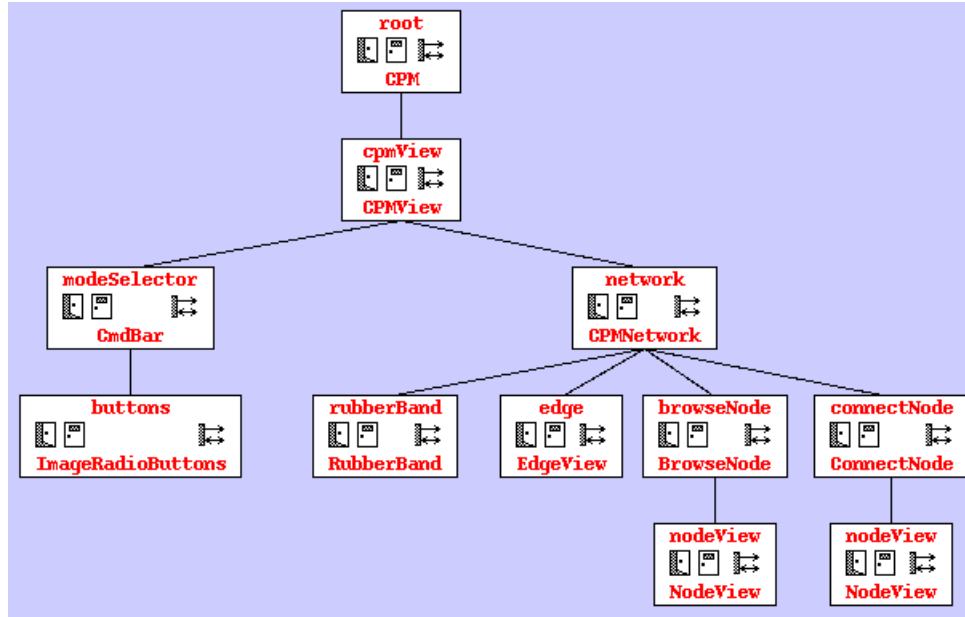
Figure 10: Skeleton architecture for the CPM planning program of figure 6.

This section presents the steps of the methodology, illustrating them with examples from the CPM planner of figure 6. These steps are summarized in reference format in appendix A.

4.1 Develop a Skeleton Architecture

The first step in the methodology is to use mockups or drawings of the user interface presentation to derive a skeleton architecture for the system. An architecture represents the components from which the system is built, and how they communicate. In Clock, similarly to InterViews [17], PAC [1], and RendezVous [14], user interface architectures are structured as a tree of components. The skeleton architecture for the CPM planner of figure 6 is shown in figure 10.

In Clock, architectures represent the compositional structure of programs. A parent component in the tree is permitted to use zero or more instances of a child component when computing its own view. For example, the *CPMNetwork* component is composed of some number of nodes, edges, and possibly a rubber band line.

Once the skeleton architecture has been identified, later steps in the methodology consist of filling in the blanks: that is, deriving the code and data represented in each component, and deriving the precise interfaces that components will use to communicate. Typically, the initial skeleton architecture will also be refined as more information is gained in later steps.

The Clock language's separation of architecture from code is crucial to this first step. The skeleton architecture can be created without having to know the details of the code contained in each architecture component, or even the details of how components communicate. Filling in the code of each component can then be done relatively independently, allowing the decomposition of the derivation process into simpler subproblems.

TASK: Connect two nodes in CPM network			
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK	CONNECTION TO COMPUTATION
Select mode *"connect"*			
~[n1] Mv		n_1!	
((~(x,y)		rubber band line follows from n_1 to *currentPos*	
) *			
(~[n']	unlocked n' : lock n'	have lock n': n' locked n'!	
[n']~	have lock n' : unlock n'	n'-! n' unlocked	
)*)*			
~[n2]	lock n_2	n_2 locked n_2!	
M^	unlock n_2	n_2 unlocked n_1-! n_2-! Remove rubber band line Show solid line from n_1 to n_2 on all users' displays	connect n_1 to n_2

Figure 11: UAN specification for the task of connecting two nodes.

4.2 Partition Interface State into ADT's

The next step in the methodology is to determine what data is required to represent the system state, and to partition this data into ADT's with well-defined interfaces. The UAN specification contains the information necesary to derive ADT's: the *interface state* column explicitly shows what state is required, and how it is modified. The *interface state*, *interface feedback* and *connection to computation* columns all make reference to system state. Therefore, by examining the state and its use in the UAN specifications, it is possible to derive ADT's and suitable interfaces.

For example, the task *"Reposition a node"* (figure 9) indicates that when a node is clicked, it becomes locked; as the node is moved, its position (*nodePos*) should be updated. When a node is displayed, its position state is used in determining where it should should be drawn. In addition to this explicit use of state, the node is implicitly enters a *depressed* state when the mouse is clicked down, and a *non-depressed* state when the mouse is released. In UAN specifications, this access to state is informal, typically written in pseudo-code (such as "$nodePos(n) := (x, y)$"), or described in prose. The process of deriving ADT's from state in UAN specifications from explicit state in UAN specifications is detailed in section 4.2.1.

Deriving ADT's represents the first step in changing point of view from the task-oriented view of the UAN to the constructional view of programs. UAN specifications implicitly contain contextual information. For example, in the task of repositioning a node (figure 9), it is implicit that the mouse motions (~[x,y]) in the second line of the chart follows clicking down on the node, and therefore should be interpreted as moving a node. In the constructional domain, in order to allow tasks to be interleaved or carried out concurrently, state must be introduced to represent such contextual
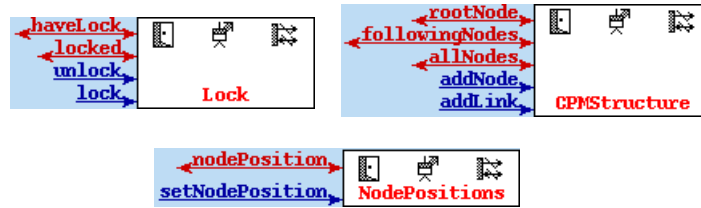
Figure 12: ADT's explicitly supporting the node connection task.

information. Section 4.2.2 shows how such implicit state is derived.

4.2.1 Explicit State

As a more complex example of how ADT's are derived from UAN specifications, consider the task of connecting two nodes (figure 11). In this task, interface state is explicitly referred to and modified. The *interface state* column queries whether a given node is unlocked, and whether the current user is the owner of the lock. Locks on nodes are set and released. The *interface feedback* column refers to the locking status of nodes, and refers to the positions of the nodes n_1 and n_2 when drawing a line between them. The *connection to computation* column modifies the structure of the network by connecting the two nodes n_1 and n_2.

The state that is explicitly used in UAN tasks must be represented in Clock via ADT's. To design these ADT's, a developer must identify what state is being used, and partition the state into logical units. Here, for example, we identify that the state can be logically partitioned into ADT's representing the network structure, the positions of the network nodes, and current locking information. Figure 12 shows the resulting ADT's and their interfaces. For example, the *NodePositions* ADT represents the positions of the nodes in the network. The request *"nodePosition n"* specifies the position of node n on the display. The update *"setNodePosition n (x,y)"* sets the position of node n to the coordinate (x, y).

Similarly, the *CPMStructure* ADT records which nodes are in the network and how they are connected. The *rootNode* request returns the root node in the network; *"followingNodes n"* returns the list of nodes connected to node n. The *"addNode n"* update adds a new node called n; *"addLink n_1 n_2"* connects nodes n_1 and n_2.

Finally, the *Lock* ADT is used to record locking of nodes on a per-user basis. The request *"locked n"* reports whether node n is locked; *"haveLock user n"* reports whether *user* holds the lock for node n. The updates *"lock user n"* and *"unlock n"* are used to obtain and release locks.

The references to state in the UAN description of figure 11 can then be expressed in terms of the updates and requests defined by these new ADT's. For example, drawing a rubber band line from n_1 to (x, y) will be implemented by drawing a line from *"nodePosition n_1"* to (x, y). The nodes n_1 and n_2 will be connected via the update *"addLink n_1 n_2"*.
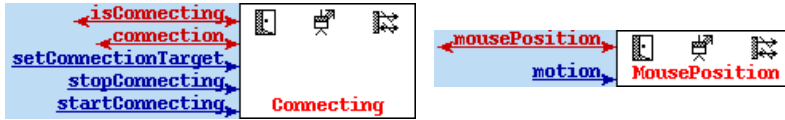
19

Figure 13: ADT's derived from the node connection task.

4.2.2   Implicit State

Derivation of ADT's from UAN specifications represents the first step in changing point of view from task-oriented specifications to implementations. Contextual information implicit in UAN charts must be made explicit in implementations where such context is not available. For example, moving the mouse pointer before the mouse button has been depressed has no observable effect. Moving the mouse after the mouse button has been depressed on a node causes a rubber band line to follow the mouse pointer. Depressing the mouse on a node therefore causes the dialogue to enter a "connecting" state, in which mouse motion is tracked by a rubber band line. The "connecting" state is exited when the mouse button is released. Such dialogue state is implicit in the sequencing notation of UAN tasks. However, to permit tasks to be interleaved or performed concurrently, Clock requires dialogue state to be encoded explicitly, using ADT's.

Figure 13 shows the *Connecting* ADT used to record dialogue state in the node connection task. The "*startConnecting* n_1" update records that the user has started connecting from node n_1. The "*setConnectionTarget* n_2" update records that the user has placed the mouse over node n_2. *stopConnecting* exits the connecting state. The *isConnecting* request determines whether the user in carrying out a connection; *connection* returns the current connection source and target.

Figure 14 shows the revised UAN for connecting two nodes, where the new updates and requests replace the earlier state references. Note that depressing the node n_1 starts the "connecting" state. Moving over a second node n_2 sets the connection target; moving away from that node resets the connection target to the *nullNode*. Releasing the mouse over a target exits the "connecting" mode, and adds a new link into the CPM structure.

In summary, this stage of the transformation requires the developer to identify all state, both explicit and implicit, used in the UAN task descriptions. From this state, the developer designs ADT's. Finally, he/she updates the UAN to use the updates and requests of the ADT's to refer to state.

4.3   Position the ADT's

Once the necessary ADT's have been defined, they must be placed in the component tree. ADT's are positioned at the lowest point in the tree where they continue to be visible to all components that use them. For example, the *NodePosition* ADT is shared by all users of the system, and must therefore appear in the root *CPM* node, where it is visible to all instances of the *CPMView*. The *Connection* ADT is shared by the children of the *CPMNetwork* component, and is therefore positioned at *CPMNetwork*. Note that at run-time, one instance of each ADT is created for each component instance. There will therefore be instance of the *NodePosition* ADT shared by all users, while there will be one instance of *Connection* per user. This means that all users see the nodes at the same position, while each user can separately carry on his/her own connection activities.

TASK: Connect two nodes in CPM network			
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK	CONN'N TO COMPUTATION
Select mode "connect"			
~[n1] Mv	startConnecting n_1	n_1!	
((~(x,y)		draw rubber band line between nodePositon n_1 and mousePosition	
) *			

(~[n']	unlocked n' :
	    lock myUserId n'
	    setConnectionTarget n'
		haveLock myUserId n':
		    n' locked
		    n'!	

[n']~	have lock n' :
	    unlock n'
	    setConnectionTarget
	        nullNode
		n'-!
		n' unlocked	

)*)*			
~[n2] Mv	lock myUserId n_2 setConnectionTarget n_2	n_2 locked n_2!	
M^	unlock n_2 stopConnecting	n_2 unlocked n_1-! n_2-! Remove rubber band line Show solid line from nodePosition n_1 to nodePosition n_2 on all users' displays	addLink n_1 n_2

Figure 14: UAN for connecting two nodes, updated to use the ADT's identified in figures 12 and 13.
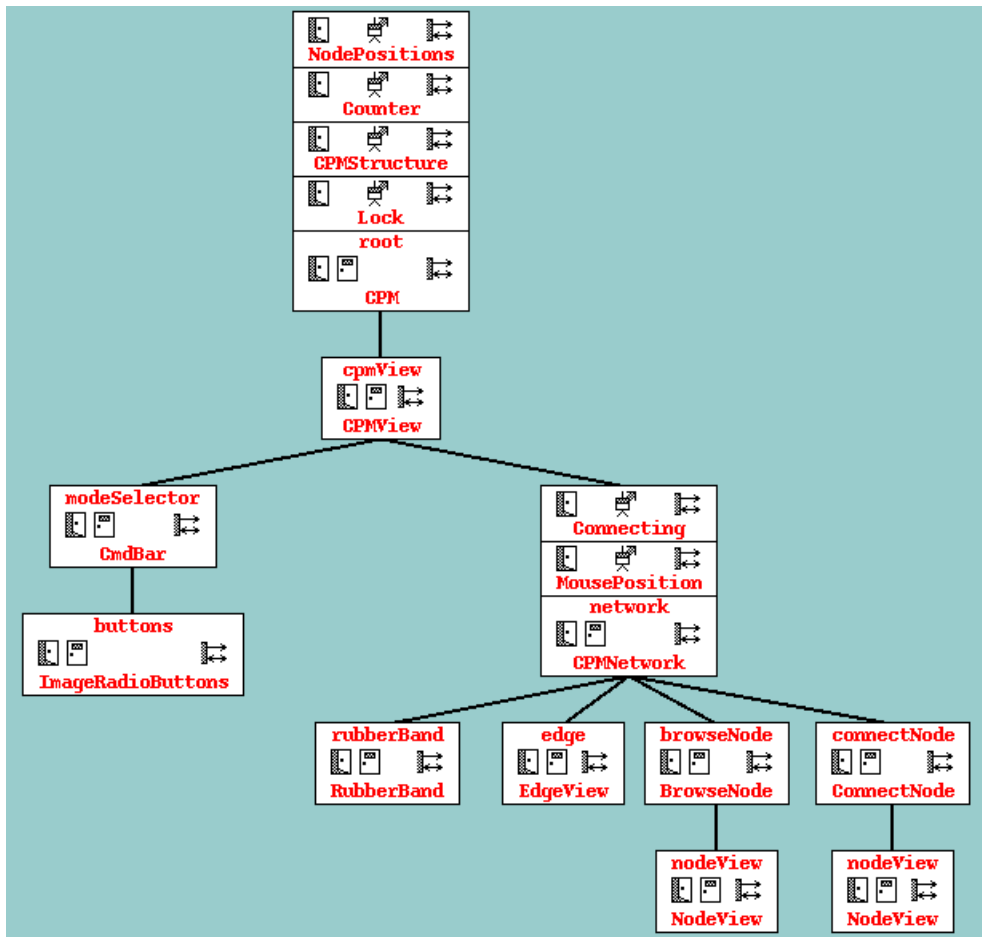
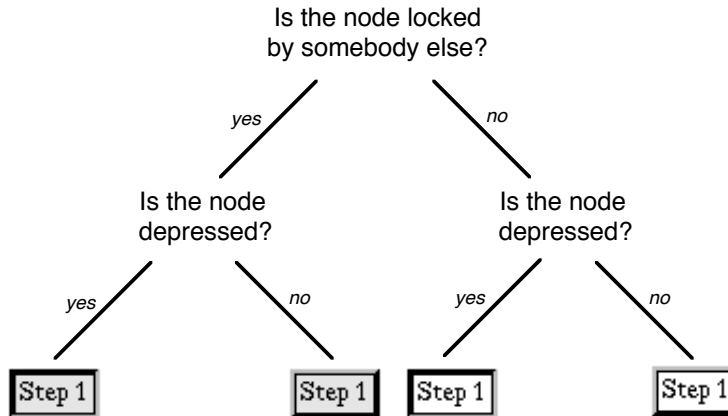Figure 15: Architecture of figure 10 once ADT's have been positioned.

Figure 16: Decision tree specifying appearance of a network node.

Figure 15 shows the final results of positioning these ADT's.

4.4 Assign Inputs

The next stage is to determine which components are responsible for handling user actions, and to annotate the components as such. For example, the *BrowseNode* component responds to mouse button clicks, while the *CPMNetwork* component responds to mouse motion.

4.5 Encode View Functions

The next major shift in point of view from task-oriented specification to implementation consists of encoding the appearance (or *view*) of the user interface on the display. In the behavioural domain of task-oriented specifications, display updates are distributed throughout the specification, appearing in the *interface feedback* column in response to user actions. This form of output specification is appropriate in the behavioural domain, since it clearly demonstrates the effects of user actions in context. The UAN's organization of output specifications is, however, less suitable for implementation: the same output actions appear repeatedly in the specification, leading to unnecessary redundancy. The close linkage of input (*user actions*) and output (*interface feedback*) leads to difficulties in maintenance. Finally, since output is stated only as a consequence of input, it is hard for programmers to be sure that every output case has been identified.

Modern languages for user interface development address these problems by separating output specifications from input specifications. This separation is realized through implicit invocation of views in response to changes in system state, either using callbacks (such as in MVC [16]) or contraints (as with Garnet [21] and similar systems.) This section shows how task-oriented output can be mapped to view specifications using Clock's constraint-based view functions.

Each component in a Clock program possesses a view function specifying its appearance on the display. View functions are specificational, in that they describe how the display is to appear independent of any input specification. The implementation of the Clock language is responsible for automatically determining when and how views are to be updated.

23

The next step in transforming UAN to Clock is to identify how Clock view functions can be used to encode the display feedback described in the UAN task descriptions. At first glance, Clock view functions would seem to be very far from UAN descriptions – view functions provide a declarative specification of a view, giving in effect a rule for how the view is to be constructed based on the current system state. These functional specifications do not explicitly refer to time – the specification holds regardless of when it is viewed. UAN specifications, on the other hand, are based on explicit user actions, in effect entirely based on time and context. It is however, quite straight-forward to translate the UAN style of display specification into the Clock style.

As an example of how to perform this transformation, we shall consider the example of how the individual nodes of a network are displayed. The basic form of a node is a box, containing some text (e.g., *"Step 1"*), with relief shading to give the illusion of three dimensions.

We first examine all UAN tasks that refer to nodes in their interface feedback. These involve the task to reposition a node (figure 9), and the task to connect two nodes (figure 14). From these tasks, we can see that there are four actions that modify the appearance of a node n:

Interface Feedback	Effect on Display
n!	n is displayed using sunken relief
n-!	n is displayed with raised relief
n locked	n is displayed with a grey background on all other users' displays
n unlocked	n is displayed with a white background on all users' displays

These display changes all correspond to changes in system state. For example, when we start connecting a node, we also depress it. Whenever we lock a node, we display it as locked on all other users' displays. By examining the locations where these changes are meant to occur, we can derive a decision tree showing how the appearance of a view can be determined from the current interface state.

The decision tree for drawing a node is shown in figure 16. The leaves of the tree show the four possible display states of a node. The node must be displayed with a grey background if it is currently locked by someone else, and must be displayed with sunken relief if it is currently depressed.

The Clock code implementing this decision tree is shown in figure 17. The basic view of a node is encoded in the *nodeBox* function, specifying that a node is a box, filled in white, surrounding text specifying the step number. The view itself consists of a *nodeView* with either a white or grey background, and either sunken or raised relief. The decisions on text colour and relief are encoded in separate functions. The structure of the view function is the same as that of the decision tree: the two decision points are encoded in the *relief* and *textColour* functions. Depending on the values of these functions, the view function may evaluate to one of the four possible presentations of the node.

Typically, deriving view functions from UAN specification is a straight-forward process. Once the UAN descriptions have been modified to access state through updates and requests (as described in section 4.2), it is usually simple to express the conditions that trigger a change in view in terms of requests. This expression then becomes a split point in the decision tree specifying the view.

The process of developing views from UAN specification gives feedback on the complexity and modality of a user interface. If it is hard to develop the condition specifying when a view has a particular presentation, it means that there is no simple rule specifying why a particular presentation may arise. This in turn might imply that the presentation is giving inadequate or misleading cues to a user of the system.

```
text = "Step " + myId.
nodeView = Box (pad 2 (Text text)).

fillColour =
    if not (locked myId)  or (haveLock myUserId myId) then
        white
    else
        % I don't have the lock
        grey90
    end if.

relief =
    if isDepressed then
        "sunken"
    else
        "raised"
    end if.

view = FillColour fillColour (Relief relief nodeView).
```

Figure 17: Code implementing the view of a network node, based on the decision tree of figure 16.

4.6   Encode Input Functions

The final stage in deriving implementations from task-oriented specifications is to encode the handling of user inputs. Most modern languages treat user inputs as events, and require the programmer to provide code mapping input events into modifications to system state. This modification is captured in UAN specifications, where inputs appearing in the *user actions* column result in state updates in the *interface state* column. For example, the chart of figure 14 specifies that clicking down on a connection node (`~[n1] Mv`) causes the node to be depressed (n_1!) and the "connecting" state to be entered (`startConnecting` n_1).

UAN specifications differ from input handling code in two respects. First, the same user action may occur more than once in the specification, whereas it must be implemented by a unique input handler. Secondly, the sequencing of user actions implicitly gives context to user actions. For example, the "`~[n]`" action (moving the mouse over a node) occurs at three places in the connection task of figure 14. This sequencing implies, for example, that we know that when the mouse is moved over the n_2 node (`~[n2]`), we must be in the "connecting" state, and that therefore some node n_1 has been specified as the connection source.

When user actions are translated to input handlers, one function must handle all cases of input being sent to a given component. This means that explicit state must be used to reconstruct the context of the user action being performed. In order to derive how input to a component should be handled, we use a decision tree similarly to how view functions are derived (section 4.5).

Figure 18 shows how moving the mouse pointer over a *ConnectNode* network node is handled. The decision tree shows how the three cases of moving over a node are distinguished. First, if no connection is being performed (as in the case of `~[n1]` at the start of the dialogue), moving over a node has no effect. Assuming a connection has been started, the effect of moving over a node is to
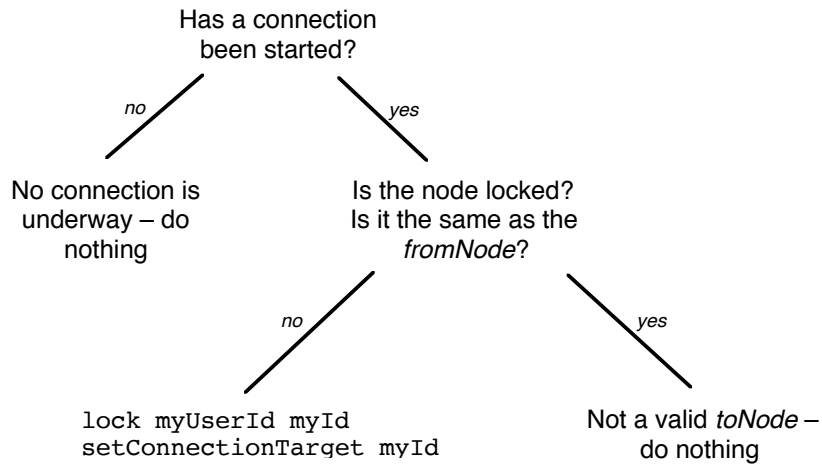
Has a connection
been started?

no   *yes*

No connection is
underway – do
nothing

Is the node locked?
Is it the same as the
fromNode?

no   *yes*

lock myUserId myId
setConnectionTarget myId

Not a valid *toNode* –
do nothing

Figure 18: Decision tree specifying how entering a network node is handled.

```
% fromNode and toNode are the          % Handle:  ~[myId]
% current ends of the connection.      enter =
fromNode = fst connection.                 if not isConnecting
toNode = snd connection.                           or myId = fromNode
                                                   or locked myId then
                                                noUpdate
% Handle:  ~[myId] Mv                      else
mouseButton "Down" =                          all [
    startConnecting myId.                          lock myUserId myId,
                                                   setConnectionTarget myId]
                                            end if.
% Handle:  M^
mouseButton "Up" =                     % Handle:  [myId]~
    if toNode = nullNode then           leave =
       stopConnecting                       if not isConnecting
    else                                            or myId = fromNode then
       all [stopConnecting,                     noUpdate
            unlock toNode,                  else
            addLink fromNode toNode]           all [
     end if.                                       unlock myId,
                                                   setConnectionTarget nullNode]
                                           end if.
```

Figure 19: Code handling the input to a *ConnectNode*. The encoding of the *enter* function is based on the decision tree of figure 18

depress and lock the node, and to enter "connecting" state. However, the node must not already be locked, and must not be the start node for the connection.

This decision tree can be easily transformed into Clock code:

```
enter =
   if not isConnecting
          or myId = fromNode
          or locked myId then
      noUpdate
   else
      all [
         lock myUserId myId,
         setConnectionTarget myId]
    end if.
```

The function *enter* is automatically invoked by the Clock runtime system whenever the mouse pointer is moved over the node. The request *myId* represents the name of this node. This function encodes that when this node is entered, as long as we have started connecting two nodes, this node is not locked and this node is not the connection source, then the node becomes locked and is set as the connection target.

Figure 19 shows the complete input code for the connection target. The *mouseButton* function handles the mouse button being depressed and released over this node. Depressing the mouse button (Mv) appears in only one context, and results in starting a connection from this node (as named by *myId*.) Releasing the mouse button (M^) can occur in two contexts – either to connect two nodes, or to cancel a connection operation. The leave update ([myId]~) occurs in two contexts – either we are not connecting, in which case there is no effect, or we were potentially connecting to this node and decided against it.

4.7   Summary of Derivation Process

This section has shown how UAN specifications help in the derivation of user interface implementations. While the derivation process has been demonstrated using the Clock language, we have argued throughout that the same techniques should be applicable using languages with similar features, in particular languages providing a strong separation of architecture from code and a clean separation of input from output.

Another feature that is highly useful is implicit concurrency control. This feature is found in some user interface languages (e.g., Sassafras [12] and RendezVous [14]), but is less common than the other features we have discussed. The next section demonstrates how the UAN helps in the specification of concurrency, and how building concurrency control into the target language greatly aids the derivation process.

5   Concurrency

Numerous researchers have discussed the difficulty of programming concurrent user interfaces [12, 13, 4, 5]. Concurrency arises in user interfaces through concurrent dialogues such as simultaneous

TASK: Create a new node		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
Select mode *"new node"*		
~[x,y] Mv	$n := nodeCounter$ $nodePos(n) := (x, y)$ $nodeCounter += 1$	Display n at (x, y) on all users' displays
M^		

Figure 20: UAN describing the task of creating a new node.

interactions of multiple users and through the concurrent playback of multimedia clips.

Concurrency is hard to implement because concurrent actions may conflict, potentially resulting in erroneous, unintuitive, or annoying system behaviour. *Concurrency control* [9] is required to restrict concurrency to avoid undesired behaviour. UAN specifications implicitly involve concurrency control. One of the great benefits of deriving Clock implementations from UAN specifications is that the Clock language also provides concurrency control, and that Clock's concurrency control rules are identical to those of UAN. These means that the difficult issues of how to arbitrate between concurrent users of a system need only be solved once, at the conveniently high level of UAN specifications. While automatic concurrency control is not yet a common feature of user interface languages, we believe that this derivation process lends convincing evidence to its importance.

Concurrency is introduced into UAN via the "||" operator. For example, the task of creating a CPM network (figure 8) specifies that creating nodes and links between nodes, and beautifying the presentation of the network are all concurrent activities, where one user can be linking two nodes, while another creates a node, and a third moves some part of the network.

The task of creating a new node (figure 20), however, shows that this concurrency cannot be un-restricted. Consider the possibility that two users create a new node at exactly the same time. A node counter is used to generate a unique identifier for each node that is created. If two nodes are created simultaneously, however, a race condition might occur, allowing both nodes to be assigned the same counter value. This would lead to the erroneous condition of two nodes being created with the same identifier.

UAN solves this problem with a simple concurrency control rule: once a sequence of user actions has been completed, the updates to *interface state*, *interface feedback* and *connection to computation* must be performed atomically. In this example, the final action is the Mv that creates the node. The UAN specification guarantees that between the time the user depresses the mouse button and the time the new node is created and displayed, no conflicting user actions may be processed. (See Hartson and Gray [10] for a formalization of the temporal aspects of the UAN.)

This rule means that if two users try to create a node simultaneously, one user will be considered, by however small a fraction of a second, to have been first. This user's node will be created and displayed, and then the second user's node will be created and displayed.

It should be noted that the UAN does not oblige an implementation to sequentialize all user actions – only those where the actions would otherwise conflict.

The need for concurrency control is also demonstrated by the task of repositioning a node. As was see in figure 9, dragging a node causes the position of the node and all attached lines to be

```
mouseButton "Down" =
   if mode = AddingNode then
      % Add a new node at this position
      let id = numstr count in
         all [
            addNode id,
            setNodePosition id mousePosition,
            incrementCount
         ]
      end let
   else
      noUpdate
   end if.

mouseButton "Up" = noUpdate.
```

Figure 21: Code handling a node creation, taken from the *CPMNetwork* component. This code implements the "add node" task of figure 20.

updated on all users' displays. The UAN's implicit concurrency control guarantees that when a node is moved, the display will be correctly updated before other users modify the positions of other nodes. This avoids the possible situation of lines being redrawn to connect to nodes that have in the meantime been moved by someone else.

This example also shows, however, that as well as being used to avoid race conditions, concurrency control is also an important part of user interface design. Consider for example, that two users try to simultaneously move the same node in opposite directions. UAN's concurrency control would guarantee a logical but annoying result – the users would engage in a tug of war, where the node would jump back and forth in response to each mouse movement. A better design (as shown in figure 9) is to allow the first user to obtain a lock on the node, move the node, and then release the lock. Other users are then forbidden from moving a node which is locked by someone else. As was shown in figure 16, we then augment the display of nodes so that a locked node appears with a grey background, making it easier for users to see where other people are working. The locking mechanism is not built in to UAN – we are simply using interface state to record who has a lock. Of course, UAN's concurrency control is being exploited to ensure that no race conditions occur in the assignment of locks.

This form of concurrency control is not to prevent race conditions or inconsistent state, but simply to provide a better user interface. It should therefore not be suprising that we have to explicitly perform node locking in our UAN specification: the UAN is not intended to arbitrate what is good user interface design, but rather to allow us to describe any design we like, good or bad.

5.1   Clock and Concurrency Control

The Clock language provides built-in concurrency control in order to shield programmers from the details of avoiding race conditions. Clock's concurrency control mechanism follows exactly the same rule as the UAN concurrency control: whenever a user input is handled, a Clock implementation must behave as if all consequent state and view updates are atomic. That is, in cases where inputs conflict, the Clock run-time system must introduce synchronization to ensure that the earlier input completes before the later input begins. Since Clock uses exactly the same concurrency control rules as UAN, mapping concurrent tasks from UAN to Clock is straightforward.

Figure 21 shows the Clock code implementing the "add node" task of figure 20. This code states that if the mouse is depressed on some empty part of the canvas in the context of the "Adding Node" mode, then a new node is to be created at that location, with a node identifier obtained via the *Count* ADT. Similarly to the UAN specification, the node is created, positioned, and the counter is incremented to provide a unique identifier for the next new node.

Clock's automatic concurrency control guarantees that race conditions cannot lead to two new nodes being assigned the same identifier. When the mouse is depressed, the system must permit all state and view updates resulting from the input to take place before any later inputs refer to the state. Therefore, when two nodes are created almost simultaneously, the later node creation is not permitted to reference the counter before it has been updated. This means that simple writing the UAN description as Clock code guarantees the same concurrent behaviour.

Similarly to the UAN, Clock allows higher-level concurrency control to be built from this primitive level. For example, as was seen in section 4.6, the UAN description of connecting two nodes (figure 14) mapped directly into the the Clock code of figure 19. The *Lock* ADT provided the locking operations developed in the UAN description. Clock's low-level concurrency control guarantees that race conditions cannot occur in the assignment of locks.

The ability to easily specify concurrent dialogues is one of the UAN's greatest strong points. Being able to solve concurrency problems at the UAN's high level and then methodically derive an implementation provides a strong incentive for creating and maintaining UAN specifications.

6   Analysis

The previous sections have presented the Clock methodology for deriving implementations from task-oriented specifications. We have argued that the derivation process helps motivate programmers to utilize UAN specifications, and helps them appreciate the value of user-centered design processes.

The main contribution of the derivation process is that it allows designers to specify the behaviour of a user interface in a task-oriented setting. Particularly with modern user interfaces, which may involve multiple users and multiple media, it is helpful to be able to design at the more natural task-oriented level. From the task-oriented specifications, the general case of view and input handling functions can be derived using the simple decision tree method shown in sections 4.5 and 4.6. This process provides a useful form of user interface evaluation: if it is hard to derive decision trees from the UAN specification, it often means that the user interface is overly complex or inconsistent.

An obvious question is whether this derivation could be performed automatically. With the UAN in its current form, automatic derivation is not possible. The UAN contains pseudo-code and prose descriptions that are too imprecise to implement without human intervention. More seriously, UAN descriptions tend to be incomplete: for example, the UAN specification of the CPM planner presented in appendix A fails to consider some cases of erroneous use. Tightening up UAN specifications to make them complete and precise risks making the notation so cumbersome that it is no longer of interest to designers.

Another approach to deriving implementations from task-oriented descriptions can be found in the area of programming by demonstration. For example, the Marquise [22] tool allows developers to demonstrate the dynamic behaviour of user interfaces. Demonstrations are in effect task-oriented, since developers walk through a sample use of the system they are creating, showing how the system is meant to respond to user inputs. From multiple demonstrations, Marquise automatically generates user interface code. Programming by demonstration approaches are highly promising, but

require further research before being practically usable. In particular, it is not yet clear how well demonstrational approaches handle concurrent user interfaces.

One of the strongest motivations for using the UAN is its high level handling of concurrency and concurrency control. It is very difficult for a programmer to anticipate and correctly handle all possible concurrent uses of a system. Even in a system as simple as the CPM planner, concurrent behaviour is complex. Consider, for example, the rules when a node may be locked: from a task-oriented point of view, it is clear that a node should be locked in two instances: when we are moving it, and between the time that we move over it as a connection target and the time we release the mouse. For a programmer to correctly isolate these cases using the implementation point of view would be challenging.

The treatment of concurrency in user interface development tools is, however, still controversial. In our Clock system, we have adopted concurrency control rules identical to those of the UAN, supporting a simple mapping from UAN specifications to Clock code. Some researchers argue that this level of concurrency control is inherently too inefficient, and advocate instead *optimistic* (or relaxed) concurrency control schemes [9, 23]. Even when the target language does not support full concurrency control, we believe it is still beneficial to use the UAN to specify what concurrent behaviour we would like to have, even if our tools do not guarantee that they will support it.

7 Conclusion

This paper has presented the Clock methodology, a process for deriving user interface implementations from task-oriented specifications in the User Action Notation. We have demonstrated the methodology by showing how an interactive system implementing a critical path method planner can be first specified in UAN, and then methodically derived from the specification.

The User Action Notation is an important tool in supporting participatory, user centered design of interactive systems. By directly aiding programmers in their implementation, the Clock methodology encourages programmers to participate in such processes.

While our presentation of the methodology is based on the Clock language for interactive system development, the methodology is primarily based on features present in many modern user interface tools: a strong separation of architecture from code; a clean separation of input and output, and high-level treatment of concurrency and concurrency control.

Acknowledgements

References

[1] Joelle Coutaz. PAC, and object-oriented model for dialog design. In *Proceedings of INTERACT'87*, pages 431–436, 1987.

[2] Herbert Damker. Spezifizierung und Architekturentwurf von Benutzungsschnittstellen: eine Fallstudie in der Clock-Methodologie. Studienarbeit, Universität Karlsruhe, September 1992.

[3] James B. Dilworth. *Production and Operations Management: Manufacturing and Services, Fifth Edition*. McGraw Hill, New York, 1993.

[4] Emden R. Ganser and John H. Reppy. A foundation for user interface construction. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 14, pages 239–260. Jones and Barlett, 1992.

[5] T.C. Nicholas Graham. Constructing user interfaces with functions and temporal constraints. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 16, pages 279–302. Jones and Bartlett, 1992.

[6] T.C. Nicholas Graham. The Clock Language. Technical Report CS-ETR-95-01, Department of Computer Science, York University, 1995.

[7] T.C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clockworks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, 1996(7):175–196, July 1996.

[8] T.C. Nicholas Graham and Tore Urnes. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'96)*. ACM Press, November 1996. (To appear).

[9] Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 207–217. ACM Press, October 1994.

[10] H. Rex Hartson and Philip D. Gray. Temporal aspects of tasks in the User Action Notation. *Human-Computer Interaction*, 7(1):1–45, 1992.

[11] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, July 1990.

[12] Ralph Hill. Supporting concurrency, communication and synchronization in human-computer interaction: the Sassafrass UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[13] Ralph D. Hill. Languages for construction of multi-user multi-media synchronous (MUMMS) applications. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 125–146. Jones and Bartlett, 1992.

[14] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous language and architecture for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

[15] Paul Hudak and Philip Wadler. Report on the functional programming language Haskell (v1.1). Technical Report YALEU/DCS/RR777, Yale University, August 1991.

[16] Glen E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[17] Mark A. Linton, John M. Vlissides, and Paul R Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[18] Rich McDaniel and Brad A. Myers. Amulet's dynamic and flexible prototype-instance object and constraint system in c++. Technical Report CMU-CS-95-176, School of Computer Science, Carnegie Mellon University, July 1995.

[19] Catherine Morton. Tool support for component-based programming. Technical Report CS-94-02, Department of Computer Science, York University, May 1994.

[20] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, School of Computer Science, Carnegie Mellon University, July 1993.

[21] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[22] Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. Marquise: Creating Complete User Interfaces by Demonstration. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, pages 293–300, 1993.

[23] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the Eigth Annual Symposium on User Interface Software and Technology (UIST'95)*, pages 111–120. ACM Press, November 1995.

[24] Jenny Preece. *Human-Computer Interaction*. Addison-Wesley, Wokingham, 1994.

[25] Richard N. Taylor, Kari A. Nies, Gregory Alan Bolcer, Craig A. MacFarlane, Kenneth M. Anderson, and Gregory F. Johnson. Chiron-1: A software architecture for user interface development, maintenance, and run-time support. *ACM Transactions on Computer-Human Interaction*, 2(2):105–144, 1995.

[26] Eric Telford. Developing a UAN browser in clockworks: a case study of incremental development using the clock methodology. Technical Report CS-96-03, Department of Computer Science, York University, June 1996.

A Collected Transformation Rules

1. Develop a skeleton architecture:

   - From the design of screen layouts, identify the compositional structure of the system, and map it to a parent-child structure.

   - Implement alternatives (e.g., alternative screens) as siblings, with a parent to choose which is active.

2. Partition interface state into ADT's:

   - Partition logically separate parts of the interface state into separate ADT's, each responsible for maintaining some part of the interface state.

   - Assignments to interface state in the *interface state* column become *updates*; references to interface state in the *interface state*, *interface feeback* and *connection to computation* columns become *requests*.

   - The collection of updates and requests handled by an ADT become the ADT's *interface*.

3. Position the ADT's:

   - Position ADT's at the lowest point in the tree where they are visible to all components that use them.

4. Assign inputs:

   - Add inputs to the interfaces of components that are responsible for parts of the interface that take user input (e.g., mouse clicks, key clicks, mouse motion).

5. Encode view functions:

   - Add a view function to each component, expressing the component's appearance as a function of the current system state. Use the design documents as a basis for specifying this presentation.

   - For each change in view expressed in the *interface feedback* column, first determine which component handles the aspect of the view that has changed. Then link the change in view to an accompanying state change.

   - Encode the view function as a decision tree, where the composition of the view depends on the current state. *if* and *case* expressions are used to select among the different possible view states.

6. Encode input functions:

   - For each type of input a component handles, provide an input function.

   - For each input specified in the *user actions* column, the corresponding input function should generate updates to effect the state changes specified in the *interface state* and *connection to computation* columns.

   - When the an input can have more than one interpretation (i.e., if the user interface is modal), each interpretation will be specified in the *user actions* column of some task. Use the interface state to distinguish between the different input interpretations. Encode these in the input function with an *if* or *case* expression.

B UAN for the CPM Example

TASK: Plan a project
USER ACTIONS
(Identify job steps and dependencies) || (Allocate resources)

TASK: Identify job steps and dependencies
USER ACTIONS
Create a CPM network

TASK: Create a CPM network
USER ACTIONS
(Specify job steps) || (Specify dependencies) || (Beautify network)

TASK: Specify job steps
USER ACTIONS
(Specify job step) +

TASK: Specify a job step
USER ACTIONS
Create a node in CPM network

TASK: Specify dependencies
USER ACTIONS
(Specify a dependency among two job steps) +

TASK: Specify a dependency among two job steps
USER ACTIONS
Connect two nodes in CPM network

TASK: Allocate resources
USER ACTIONS
Perform initial allocation
(Adjust allocation) *

TASK: Perform initial allocation
USER ACTIONS
Allocate resources to job steps

TASK: Allocate resources to job steps
USER ACTIONS
(Allocate resources to a job step) *

TASK: Create a new node		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
Select mode *"new node"*		
~[x,y] Mv	$n := nodeCounter$ $nodePos(n) := (x,y)$ $nodeCounter += 1$	Display n at (x,y) on all users' displays
M^		

TASK: Select mode (mode)		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
{ ~[mode] Mv		*mode*!
M^	$currentMode := mode$	*mode*-! $\forall m'.m'!! : m'-!!$ *mode*!!
}		

TASK: Beautify network
USER ACTIONS
Reposition nodes

TASK: Reposition nodes
USER ACTIONS
(Reposition a node) *

TASK: Reposition a node		
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK
select mode *"move"*		
`~[n] Mv`	lock n	n! n locked
`( ~[x,y]`	$nodePos(n) := (x, y)$	display n at (x,y) on all users' displays, moving all connected lines and rubber-band lines.
`)* M^`	unlock n	n-! n unlocked

TASK: Connect two nodes in CPM network			
USER ACTIONS	INTERFACE STATE	INTERFACE FEEDBACK	CONNECTION TO COMPUTATION
Select mode *"connect"*			
`~[n1] Mv`		n_1!	
`(( ~(x,y)`		rubber band line follows from n_1 to *currentPos*	
`) *`			
`( ~[n']`	unlocked n' : lock n'	have lock n': n' locked n'!	
`[n']~`	have lock n' : unlock n'	n'-! n' unlocked	
`)* )*`			
`~[n2]`	lock n_2	n_2 locked n_2!	
`M^`	unlock n_2	n_2 unlocked n_1-! n_2-! Remove rubber band line Show solid line from n_1 to n_2 on all users' displays	connect n_1 to n_2