York University
Department of Computer Science

Technical Report CS-96-03

Developing a UAN Browser in ClockWorks: a case study of incremental development using the Clock methodology

---

Eric Telford
June 19, 1996

Abstract

The User Action Notation (UAN) is a notation for specifying how a user performs tasks using a given interactive software system. An accurate and complete UAN specification provides a clear description of what tasks the user needs to perform to accomplish his goals, and how the user will interact with the system to accomplish those tasks. This provides the software developer with a clear direction for implementing the system's behaviour, and a reference against which the success of the system design and implementation can be measured.

However, it is the exception, rather than the rule, that a thorough and accurate specification of the tasks the user needs to perform is completed before implementation work begins. The reality of interactive software development is that specification and implementation are not chronologically distinct stages. Rather, they are interleaved, as the developer moves back and forth between them, incrementally refining both the system implementation and the system design. If the UAN specification of the system evolves along with the implementation and design, the specification continues to serve a vital role, ensuring that the evolving system continues to meet the needs of the user.

Unfortunately, due to the structure of the notation, it is hard to edit or even read a complex UAN specification using standard text-processing tools. Because UAN specifications are hard to maintain, they tend to be abandoned during the incremental system development process, or simply left to the end of the process and used as a documentation tool.

This report documents my experience developing a UAN browsing tool using the ClockWorks development environment. ClockWorks is designed to support the incremental method of development discussed above, which I refer to as the "lazy programmer" method. The report contains a critique of how well ClockWorks supports this method, and shows how the use of UAN can be incorporated in the specification/ implementation cycle.

TABLE OF CONTENTS

1 Introduction

This report documents my experience developing a User Action Notation (UAN) [1] browsing tool using the ClockWorks [2] development environment. The goal of the project is to provide a documented case study of ClockWorks in action, with a view to analyzing how fully ClockWorks achieves its aim of supporting the incremental application development style described in this report. The choice to develop a tool for the UAN in the case study is motivated by the belief that UAN specifications can and should play a vital role in the incremental application development process, but that currently, due to the difficulty of maintaining UAN specifications with existing tools, their potential remains unrealized.

Section Two deals with the UAN, providing an overview of the notation and the problems it poses, and then advancing arguments for the value of sophisticated UAN tools.

Section Three deals with incremental development, providing a description of the incremental development style (referred to as the "lazy programmer" style). It also describes the Clock methodology [3] [4] [5] (on which ClockWorks is based) and describes how Clock is intended to support incremental development. The role that UAN can play in the incremental process is addressed, and the current work exploring the close relationship between the UAN and Clock is mentioned.

Section Four details my actual development experience during the project, highlighting the way in which I followed the "lazy programmer" method. Three main iterations in the implementation/specification cycle are documented, and the role of UAN in the process is described. This section also contains a description of the state of the developed browser software at the end of the project along with suggestions for future work.

Section Five contains an analysis of how well Clock and ClockWorks do in fact support the incremental process, discussing the conflict between the desire for development speed and control of application structure, and the value of the visual ClockWorks tool.

Section Six concludes the case study.

2 The User Action Notation

2.1 Overview of the UAN

The User Action Notation (UAN), developed by Hartson, Siochi and Hix, is a notation for specifying how a user performs tasks using a given interactive software system. An accurate and complete UAN specification provides a clear description of what tasks the user needs to perform to accomplish his goals, and how the user will interact with the system to accomplish those tasks.

A UAN specification consists of a set of tables, each table describing a task that a user may perform while using the system. For example, in a disk-file management application, there might be tables describing tasks such as "Move File", "Select Files", "Delete Files", and so on. The tables have the format shown in figure 1. The table's title is on top, with from one to four columns underneath. (Empty columns may or may not be shown.) The columns are titled "USER ACTION", "INTERFACE FEEDBACK", "INTERFACE STATE" and "CONNECTION TO COMPUTATION". Information on the same row across the columns of the table is considered to 'happen' at the same time, so a user action could

immediately cause a change in interface feedback, state, or a computation call. Each column deals with a different aspect of the specification.

TASK: Delete File			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
~ [file_icon] Mv	file_icon-!: file_icon!, ∀ file_icon'!: file_icon-!	selected = file	
~[x,y]*	outline(file_icon) >~		
~[trash_icon]	outline(file_icon) >~, trash_icon!		
M^	erase(file_icon), trash_icon!!	selected = null	mark file for deletion

Figure 1: A UAN Task Table for deleting a Macintosh file. Taken from [1]

The USER ACTION column contains a list of actions that the user performs to complete the task described by the table. Each of these actions is either an 'atomic' user action (such as a mouse click), or a more complex subtask (such as "Delete File", which, for example, could be a subtask of the task "Clean up desktop") that has its own table. In the latter case, the action is performed by performing the actions in the subtask's table (analogous to a subroutine call). In this report, atomic actions are referred to as "terminal" tasks, and subtasks representing links to other tables as "non-terminal" tasks.

The actions and subtasks in the USER ACTION column may be sequential, interleaved, optional, concurrent, repeated, and so forth. The UAN provides a rich set of additional symbols that allow the UAN user to specify the relationship between subtasks. In figure 1, for example, the notation ~[x,y]* indicates that the user can move (~) the mouse pointer to any screen location ([x,y]) zero or more times (* indicates Kleene closure); i.e. while holding down the mouse button, the user can drag the file icon outline around the screen indefinitely. If no special symbols are present, a sequential ordering of tasks from the top to the bottom of the column is assumed. A full description of the UAN symbol set is beyond the scope of this report. Interested readers are directed to [1].

The INTERFACE FEEDBACK column contains descriptions of how the system responds to user actions. For example, to indicate that a file icon should be highlighted (UAN uses the symbol '!' to indicate a 'highlight' state) when clicked on with the mouse, the UAN user writes 'file_icon!' in the INTERFACE FEEDBACK column beside the USER ACTION entry 'Mv', the UAN symbol for depressing the mouse button.

The INTERFACE STATE column is used to track changes in state-related information. For example, it might be desirable to track the name of the currently selected file in the variable *selected*. If a user operation with the mouse causes a new file icon to be highlighted, *selected* is assigned the new file name (selected = file) in the INTERFACE STATE column directly to the right of the Mv action.

Finally, the CONNECTION TO COMPUTATION column is used to specify calls to non-interface components of the application. For example, placing a file icon in the trash and releasing the mouse button triggers the system to mark the file for deletion.

The tables in a UAN specification are related to one another in the form of a directed graph. The graph contains the main task (i.e. the overall goal of the system user) as the sole 'source' node in the graph (e.g. 'Put on shoes': see figure 2), some set of middle nodes (those that lead to other tables as well as possibly containing terminal tasks, e.g. 'Put on left shoe'), and another set of 'sink' nodes (those tables that consist solely of terminal tasks, e.g. `Tie laces'). The graph is directed if the edges are viewed as representing the relationship "is accomplished by"; if 'Put on shoes' is accomplished by 'Put on left shoe', the relationship is represented with an arrow from the former task to the latter.

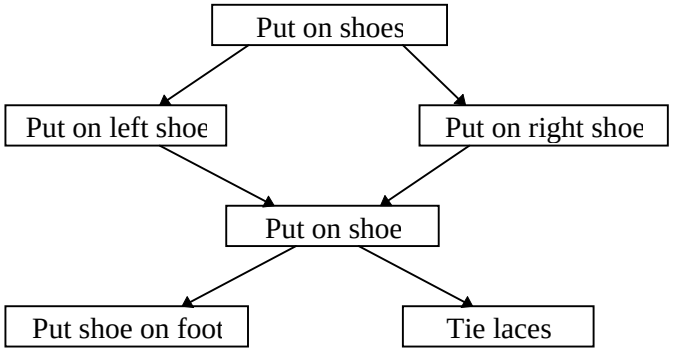

Figure 2: A UAN Directed Graph

The task hierarchy can be (and often is) viewed as a tree by duplicating shared subtasks. In a tree representation, it is important to remember that each task is defined solely in terms of its subtask structure, and is not defined in terms of its position in the hierarchy (referred to as the task "context"). The subtask 'Put on shoe' required to accomplish 'Put on right shoe' is identical to the 'Put on shoe' required to accomplish 'Put on left shoe'. To avoid confusion, and to emphasize shared subtasks, a directed graph representation of the hierarchy seems preferable to a tree representation. In the case of a very complex task hierarchy with many shared subtasks, however, a directed graph would become hard to read and the tree representation would be preferable.

The UAN is a very powerful and precise notation. In theory, a complete UAN specification can be handed to a programmer for implementation. Unfortunately, developing a complete UAN specification is extremely difficult.

2.2 Problems with the UAN

The most common complaint heard about the User Action Notation is how difficult it is to use. Most of this difficulty comes from the attempt to force what is essentially a directed graph structure into a linear form, a process which is forced on the UAN user who has only a word processing tool at his disposal.

Given the complex internal structure of a UAN specification, it's not surprising that forcing it into a linear format makes it difficult to use or even understand, since the person writing the specification is forced to arbitrarily sort the tables. In a complex specification, it is very hard (if not impossible) to keep all of the related tables adjacent to one another. This results in: endless searching back and forth through pages of specification to trace the edges of the graph from task to task; the frequent inability to view related tasks at

the same time without disordering the document; a lack of easily identifiable information about task contexts; and so forth. A UAN specification with the table structure described by Hartson et al., while a powerful and useful notation, cannot be represented in a linear format without obscuring a lot of the specification's information.

In order to make the full amount of information contained in a UAN specification available to the user, one must either provide extensions to the definition of UAN, or provide more sophisticated software tools that can extract implicit information from a specification and present it conveniently to the user.

It is apparent that in the original definition of UAN, the focus was on providing a notation to describe the low-level atomic user actions required to perform specific tasks. It is possible to concisely define low-level tasks entirely within a single table, and tables containing only atomic actions are relatively independent of other tables and can reasonably be presented linearly. In a complex specification, however, this type of table makes up a diminishing percentage of the overall hierarchy, and does not provide any of the vital higher level understanding of the task hierarchy structure. Much of a large specification consists of highly interconnected tables, and it is in the documentation of the relationships between task tables where most of the difficulties with UAN lie. If UAN is to become useful in larger projects, its reliance on table interconnections must be addressed, and possibly incorporated into the notation. One suggested extension to the notation would be the inclusion of context information with each table, perhaps by adding table rows identifying the various ancestors of the table in the task hierarchy.

The disadvantage of extending the notation is increased complexity. Careful work would be required to balance the task of making implicit specification information explicit with the task of maintaining a readable notation format.

The second option, providing more sophisticated tool support for the notation as it currently exists, is investigated in the following section.

2.3 Providing UAN tool support

As mentioned, the current definition of UAN does create specifications that contain useful information for the system developer. The problem is that much of that information is in a form difficult for the UAN user to obtain using a standard text processor. Therefore, the idea of developing more intelligent software to help the user manage specification information is appealing.

The first step in developing software to support the use of UAN is to decide exactly what tasks are faced by a UAN specification user: that is, develop a UAN specification! I limit the following discussion to the tasks faced by someone attempting to understand and use an existing, static specification; the tasks faced by someone editing a UAN specification is a much larger superset of these.

Browsing a UAN Specification can be described as a repetition of three primary tasks:

1. *Viewing the details (i.e. table) of a specific task.* In this task, one knows the name of the task, and has to find and display the table associated with that task (I am, of course, assuming a non-terminal task) so that one can view the actions/subtasks required to accomplish it, and identify features of the task specification such as critical (i.e. mandatory) subtasks, repetitive subtasks, etc.

2. *Viewing the context of a specific task*. This refers to identifying where in the task hierarchy a specific task may be found, and its relationship with other tasks, i.e. identifying its ancestors and descendants. Note that one may wish to concurrently view the context of many different tasks, or even many different contexts of the same task.

3. *Viewing a section of the Task Hierarchy*: This refers to identifying properties of the task hierarchy itself, such as identifying complex tasks (those with many subtasks), determining the hierarchy depth (perhaps the specification is too complex or not complex enough), identifying tasks that show up often in the hierarchy (these tasks need special attention, since they will be performed often), as well as giving a quick overview of the hierarchy structure. One may wish to see all or just part of the hierarchy, ideally with a variety of levels of detail.

The following UAN table describes these tasks. See Appendix A for the complete UAN specification of the browser tool developed during the course of the project.

TASK: Browse UAN Specification			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
({ View Table of Specific Task }			
⇔			
{ View Context(s) of Specific Task(s) in Task Hierarchy }			
⇔			
{ View Section of Task Hierarchy }			
)+			

Figure 3: Top Level Specification of a UAN Browser

In figure 3, the '⇔' symbol indicates task interleaving: each of the three main tasks may be interrupted at any point by one of the others, and returned to subsequently. The braces indicate that each of the three tasks is optional: a successful 'browse' session may or may not include the task. The parentheses and '+' symbol indicate that browsing is repetitive: any or all of the tasks may be performed any number of times. One uses '+' rather than '*' closure to highlight the fact that a browse will involve at least one of these tasks being performed at least once.

2.3.1 Browsing a specification in a completed Browser

In theory, all of these tasks can be directly supported by a fully functional graph-structured browser (the initial development target for this project). The tool could provide a graphical view of the task hierarchy which would allow users to view and query the task hierarchy directly. For example, the user could ask that all instances of a specific named task be highlighted, or that all tasks modifying the interface state variable 'foobar' be highlighted. Tasks in the hierarchy could be expanded to show their table details, and tasks could be traced via the table text or the graph representation. Multiple tables could be viewed concurrently, allowing the user to compare task contexts and task structures, perhaps identifying common subtask

sequences. Finally, the tool could provide a set of dialogs permitting the user to identify implicit or non-visual properties of the specification, such as the average number of steps per task, or even generate information, like a sample sequence of tasks to be performed in a testing session. The amount of information in a UAN specification is large; an appropriate tool that retains the "three-dimensional" aspects of the information will make the UAN user's job much easier.

2.3.2 Browsing a specification in a word-processor environment

In fact, of the three main subtasks indicated above, only the first, "View table of specific task" can be reasonably performed. Given the name of a task, the user can search the document for the table with that title, or use an index. However, without additional information, the context(s) of tasks is hard to determine, often requiring multiple searches of the document, since no "ancestor" data is included with each table. Tracing back several ancestors is such a complex cognitive task that the user often forgets part way through which ancestor he or she wanted to find, and why. The last task, viewing a section of the hierarchy, is impossible without a pencil and sketch pad.

While all of the information required to complete the second two tasks is present in the specification, it is in an essentially unusable form, and therefore severely hinders the UAN user.

3 Incremental Development, ClockWorks, and UAN

3.1 The Lazy Programmer

In classic system design theory, the system development process goes through one or more distinct and well-defined stages of specification, followed by an implementation stage. It is expected that a large percentage of the system (both the tasks that the system will support and how it will support them) is well defined and understood before implementation begins. In the interactive software domain, however, it has become increasingly recognized that it is the exception, rather than the rule, that a complete and accurate system specification is completed before implementation work begins.

In fact, even viewing specification and implementation as chronologically distinct stages doesn't reflect reality. Many interactive software developers begin implementation with a vague and incomplete system specification, entering a cycle of 'getting it wrong', respecifying part of the system, and reimplementing. Only gradually, correcting as they go, do they develop what is actually needed and useful. I refer to this as the "lazy programmer" style.

The lazy programmer style works as follows:

- Sit down with the system users several times and talk about the system: what do they want, what do they need? Clarify some terms, get a "general" sense of the system. A picture develops in your mind, both of the externals (layouts) and the internal structures that will form the system. Make plenty of sketches on paper. If you're lucky, you are working in a group, and can debate your perceptions of what is required with the other group members.

- Unable to think of any other useful (but dull) preparatory work to do, decide to use your chosen interactive system builder as a specification aid, i.e. see if you can implement what you understand so

far, and use the experience to increase your understanding of the system. Sit down at a computer somewhere and program as much of the system as you can until you encounter the glaring omissions and inconsistencies in the current specification.

- Repeat, *not necessarily in order,* until satisfied:

    1. Sit down somewhere else and try to sort out the inconsistencies and flaws in part or all of your last specification. Depending on the quality of the development environment, some of the problems with the specification may not be inherent, but caused by the specification not meeting the development tool's requirements. Come up with a specification that might run and solves at least one of the problems you've identified. Take things one step at a time. Return to the computer.

    2. Implement changes to the system structure: each of these changes will destroy a percentage of the previous work and probably "break" the system. If you are cautious and lucky, each change can be made separately, and the system made workable between each change. The more likely scenario is that a single structural change will side-effect the entire architecture and force you to re-write a large portion of the internals before system function is restored. This step includes potentially gruesome debugging due to the number of changes made, and the possibility of extensive side-effects caused by the changes.

    3. For a break from heavy thinking at any point while programming, decide on changes to the "look and feel" of the system, i.e. changes to colours, fonts, object layouts, etc.

    4. Implement the changes to look and feel: with modern interactive tools, these changes will probably not "break" the system, and each change in specification is probably rapidly implemented. Debugging these changes is usually straightforward.

    5. Extend existing (or add new) system components, implementing new system functionality. This includes both new functionality permitted by changes made to the system internals in step 2, and functionality previously implementable, but not implemented due to awareness of the internal structure's flaws. The debugging for this step is usually straightforward, since new functionality can be tested in small pieces as it is added to an already working system.

    6. Realize, either on your own or through user feedback, that the current specification as implemented (perhaps due to contortions to get it to "run") cannot perform some vital task.

3.1.1 Clock, ClockWorks and the Lazy Programmer

Clock is a component-based architecture language derived from Haskell [6]. It is designed to facilitate the rapid prototyping of graphical user interfaces by providing built-in graphical primitives and high level graphical abstractions that permit the programmer to bypass much of the pain of low-level graphical programming. It is therefore also designed to support the lazy programmer style of iterative refinement by making changes to applications quick and easy to perform.

Clock provides a simple but powerful graphical primitive called a DisplayView. A DisplayView represents some visual element of the Clock application, for example a box or a line of text. Simple DisplayViews can

be extended indefinitely, providing enormous flexibility. For example, a DisplayView containing a line of text can be included within the constructor for a Box DisplayView, providing a boxed text element. Fonts, colours, line styles, screen positions, etc. can all be specified by simply composing the desired DisplayViews. Multiple DisplayViews can be grouped together into a single higher-level DisplayView, either with explicit positions or with positions relative to one another (e.g. above or beside). For example, a DisplayView representing a list of files might be composed of a set of boxed-text DisplayViews (one per file) arranged above one another.

This structuring of DisplayViews within DisplayViews leads naturally to a tree-like application hierarchy. The entire application is represented as an abstract DisplayView that contains, in some relationship to each other, the set of major visual components (also DisplayViews) which in turn are composed of lower-level DisplayViews, and so on down to the smallest visual elements of the application, such as buttons and text boxes. The tree structure of a Clock application can be seen in figure 4, which shows a sample of the UAN Browser application.

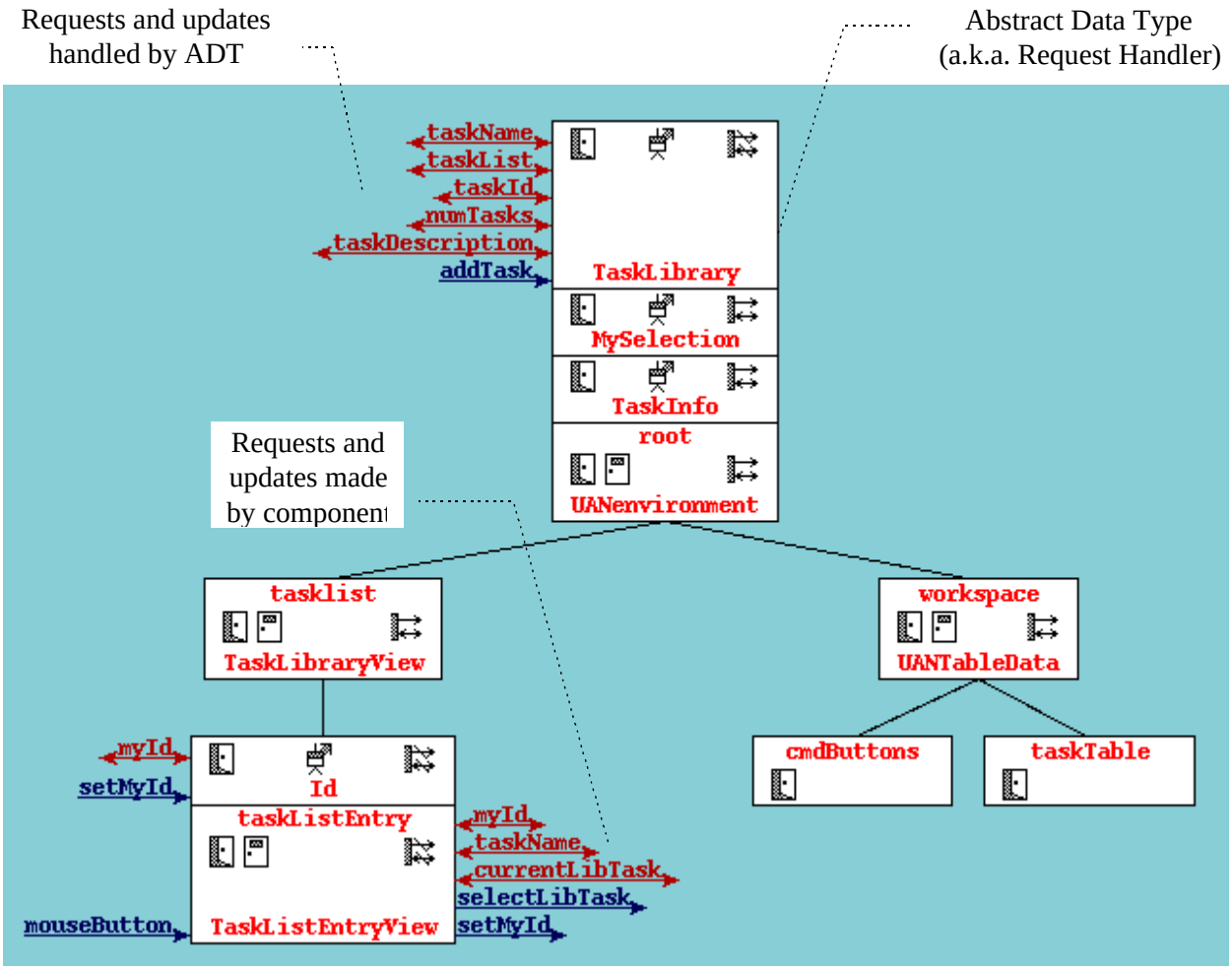

Figure 4: A Sample Clock Architecture.

Figure 4 is a screen capture taken from ClockWorks. ClockWorks is a development environment tool that permits the programmer to directly manipulate Clock architectures. As can be seen from figure 4, the tree

structure is divided into blocks, or components. Components are used to provide a means of dividing the elements of an application into logically related groups or classes: in figure 4, for example, information about UAN tables is grouped in the taskTable component, and command button information in the cmdButtons component. Both of these are subordinate to the workspace component, which provides the window in which they are displayed and manipulated.

Each component contains a single *view* function which defines its DisplayView. Of course, the component's view statement may contain DisplayViews of sub-components, permitting the DisplayView hierarchy that defines the application to be built.

Each component in a Clock architecture can also be associated with some number of "Abstract Data Types", such as the 'Id' and 'TaskInfo' ADTs shown in figure 4. These provide state information (i.e. variables) for the application which can be used to provide feedback to the DisplayViews, changing the visual appearance of the application appropriately as the user changes the state.

Programmer-defined messages are passed *up* the architecture hierarchy only, in the form of updates and requests. Updates are messages used to alter the system state, while requests are messages used to query some aspect of the current state. (In figure 4, 'TaskName' is an example of a request; 'setMyId' is an update.) When a component makes a request or update, the message is passed up the hierarchy until a component or abstract data type is encountered that handles the message. User input is handled by trapping updates such as MouseButton events in low-level components, and passing any necessary messages up the tree to accomplish the state changes specified for the user's action. To provide feedback to the user that the system state has changed, the visual display of a Clock application is recalculated automatically when an update changes some part of the system state associated with a visual element, such as an (X,Y) window position coordinate variable.

Clock and ClockWorks are designed to make the behavioural specification of the system a high-level task. The programmer specifies only what the system should do, and how the parts of the system should interact with one another. He does not have to concern himself with how to implement the behaviour. The result is a very fast specification method, which supports the Lazy programmer approach by making changes rapid and mistakes easy to recover from. The visual representation of the architecture in ClockWorks provides complexity management, direct manipulation of elements and memory prompting, further speeding the task of reworking and refining an application's architecture.

3.1.2 UAN and the Lazy Programmer

If at any point in the development iterations a lazy programmer could press a button and have an up-to-date UAN specification magically appear, no doubt UAN would be a lot more popular than it is. Unfortunately, a UAN specification can't be generated from a system specification, since the latter describes what the various elements of a system do and how they are currently related to one another, while UAN specifies why the various elements of a system are required, and how they need to be combined to accomplish the system's tasks. This different point of view is why UAN specifications, if they could only be kept up to date, would be so helpful to the lazy programmer.

It is often assumed that developers know what they want to implement, but have to try many times to figure out how to do it. This is not true for the lazy programmer. As outlined above, the developer's understanding of *what* the system needs to be able to do is as vague at the start as his understanding of how

to implement what he is sure of. As with the 'how', the 'what' of a system is clarified slowly, through iterations of development. It is more costly, however, to make mistakes about what a system should do than how it should do it. If a developer is unclear on some part of the user's required tasks, he may spend a large amount of time implementing unnecessary functionality. On the other hand, even if the programmer implements some needed function in the worst possible way, the system has still gained in usefulness. The use of UAN to specify the major tasks and subtasks required can help the programmer to clarify what needs to be done, and avoid expensive misunderstandings.

Even at the "how" level, an up-to-date UAN specification can provide a programmer with valuable contextual information. The normal programming process of working directly on the system's behaviour often doesn't take into account the total context in which the behaviour is being specified. The programmer may implement a brilliant solution for one task, only to realize later that his solution doesn't work at all well for a different, but related task that he hadn't identified. The programmer can then either rewrite the code to handle both tasks, or create a new element to handle the newly identified task. In the first case, time is wasted; in the second, the system implementation becomes more complex.

These uses of UAN are predicated on accommodating the laziness of the lazy programmer: if a UAN specification of the system is to evolve along with the implementation and design, it must be as easy to update as the system specification. Ideally, tools would be available to allow the user to check his system specification against the task specification, bringing the UAN directly into the implementation process. Unfortunately, as discussed above, it is hard to edit or even read a complex UAN specification using standard text-processing tools. Because of this, UAN specifications tend to be abandoned during the incremental development process, or simply left to the end of the process and used as a documentation tool. This leaves the lazy programmer with no easy way of documenting what tasks the growing system is able to, or has yet to, support, as his understanding of task requirements grows through the iterations of development.

3.2 Mapping specification to implementation: UAN and Clock

It is apparent that Clock and UAN have the potential to exist as complementary tools within the lazy programmer paradigm. Clock (and ClockWorks) provide a means to rapidly create and modify the behavioural specification (i.e. implementation) of a system, while UAN, with proper tool support, provides a means to rapidly create and modify task-oriented specifications. With both sets of tools available, the lazy programmer would be able to more easily perform the switch back and forth between implementation and specification that occurs so frequently during development iterations.

In fact, in [4] Graham et al. explore the possibility of exploiting direct similarities between the task-oriented UAN specification and the high-level system-centred specification used by Clock. In the paper, he demonstrates a method of translation from a UAN to a Clock specification, mechanically linking task specification to behavioural specification.

If this translation method proves complete, it means that UAN can be used as a direct link in the specification-to-implementation process. An obvious precondition of this use, of course, is the provision of more sophisticated UAN tool support. Once the tools exist, it becomes possible to envision tool-assisted conversion from UAN specification to an implementable system-centred specification, and the corresponding jump in the realization of UAN's potential.

4 Developing a Clock application: a case study

This section of the report focuses on the actual experience of developing the UAN browser application in the ClockWorks environment. This section therefore consists of anecdotal information, observations, and suggestions. The information presented is informal; this was not a controlled experiment, and the observations and suggestions do not constitute an objective comparison of the environment with any specific control group. Rather, it is hoped that this information will provide as complete a subjective evaluation as possible.

At the end of the experience, I can identify two fundamental reactions to ClockWorks. The first is my opinion that the potential of the ClockWorks environment for combining rapidity of development with maintenance of coherent internal system structure is unequaled by any other interactive system tool that I have used to date. The second is the nagging fear that *any* attempt to enforce a specific internal structure will lead to rejection: fundamental to the success of all of the visual tools to date has been the complete freedom of internal architecture that they permit. The bias of industry and programmers against imposed styles, even general architectural styles, is broad and deep.

The experience has also deepened my belief that the usefulness of UAN specifications is tied to their ability to evolve concurrently with the system specification. This project started with task specification (although informal) and ended with further task specification once the initial tasks had been implemented. The further task specification set the direction for the next phase of implementation, and this back-and-forth would have continued if the project had not ended. In this project, the UAN showed a glimpse of its potential usefulness. If convenient tools for UAN manipulation existed within or alongside ClockWorks, the back-and-forth between task and system behaviour specification could occur more frequently. It is my belief that the more frequently the developer returns to the identification of the user's tasks, the less likely developer time will be wasted implementing useless functionality.

4.1 The experience

True to lazy programmer form, I began the implementation of the browser with an incomplete and unclear idea of what the browser needed to be able to do. This was normal: not only was I learning what the browser needed, I was also learning the entire ClockWorks environment, based on a declarative programming paradigm that I had never used before. Intuitively, it felt right to "play" with the system, implementing some simple and basic browser function to begin with. It didn't appear necessary to know right at the start about everything the browser had to be able to do.

Some initial specification work had to be done, of course. I had to identify a user task that the system would support: I identified the most obvious of the tasks mentioned in section 2.3: "View table of specific task". To perform this, I realized that the user would need to navigate up and down the task hierarchy, display multiple tables concurrently, and move and hide tables.

I also had to decide on an approach to displaying the information to the user. The first approach that suggested itself was that used by ClockWorks: a tree display on a scrollable work surface. The obvious drawback of this approach was the complexity of the visual elements: specifying a system that could neatly display arbitrary graph structures seemed a large task. The second approach that I looked at was one used by Borland's C++ Object Inspector utility[7]. This utility allows the user to browse C++ objects using a

series of free-floating windows containing both passive information about the object and active links to any sub-objects (classes, arrays, etc.) the object contains. Clicking on an active link pops up another free-floating window with a detailed view of the sub-object's information and summary information on the parent object. This second approach seemed to provide all of the required navigational function if I included a list of the ancestors of a table in the window displaying it (referred to as the table 'stack'). Further, the free-floating windows of the second approach seemed easier to specify. This approach was chosen.

All of this work was done in my head and through rough sketches on paper. I didn't make any use of UAN at this point, because:

- it was a nuisance to write UAN specifications using emacs or vi, the only available editors on the computers running ClockWorks
- I already felt I had a clear understanding of how I wished to accomplish the 'view table' task, so the UAN would just be documentation
- not knowing ClockWorks' capabilities, I didn't want to spend the considerable time required to specify the task in detail until I was sure that I could implement it as I desired.

Once the initial approach was decided, development of the browser during the project spanned three iterations of the lazy programmer cycle. The majority of my time was spent specifying and respecifying the system behaviour in order to end up with a system that would acceptably support the view table task.

Once the 'view table' task had reached an (arbitrarily-determined) acceptable state and the dust had settled, I became interested in looking at what *else* the system should do. At this point I developed a formal UAN specification for the browser, which quickly showed me the next steps to be taken.

The following three sections detail each major iteration of the lazy programmer cycle that I followed during the project.

4.1.1 First iteration

4.1.1.1 The Details

Once the visual paradigm was established (the 'stack' approach discussed above), the initial specification of the major system components was rapid. It was obvious that a 'desktop' containing the tables would be required. The contents of each table would be some (possibly empty) set of stack items, the table's title, the UAN column headings, and rows of terminal and non-terminal subtasks. To simplify matters, the association of each terminal task with 'interface feedback', 'interface state', and 'connection to computation' data was left for later. (It was not addressed until the very end of development: fortunately, it was trivial.) Figure 5 shows the browser's ClockWorks architecture diagram during the first iteration.

Most of the difficulties I experienced in this iteration were caused by unfamiliarity with ClockWorks. I had some problems executing the architecture (segmentation violations), and a long period where nothing would come up on the display except a small empty X window, followed by a massive degradation in X performance until the process controlling the window had been killed. This problem was always traceable in the end to a syntactic or type error on my part, usually an error in a DisplayView grouping (e.g. having a value that evaluated to a DisplayView list rather than a DisplayView as a member of an 'above' list).

By the end of this iteration, I reached the point where I could display multiple tables by clicking on any non-terminal subtask (named ActionViews during the first iteration) of a displayed table to display the table with the same name as the subtask. I was intending to implement table motion (dragging a table around the desktop), and table layering (bringing a table to the front by selecting it, to allow overlapping tables), when I realized that my internal structure had a deficiency that would potentially cause errors in ClockWorks. The problem lay in the TaskInfo ADT connected to the DeskTop component (see figure 6: details of lengthy updates are elided). TaskInfo consisted of a list of dictionary entries representing all of the non-terminal tasks in the system. Each entry consisted of the non-terminal's task identifier (a string) as a key followed by the list of subtasks (again identified by strings) needed to define the task.

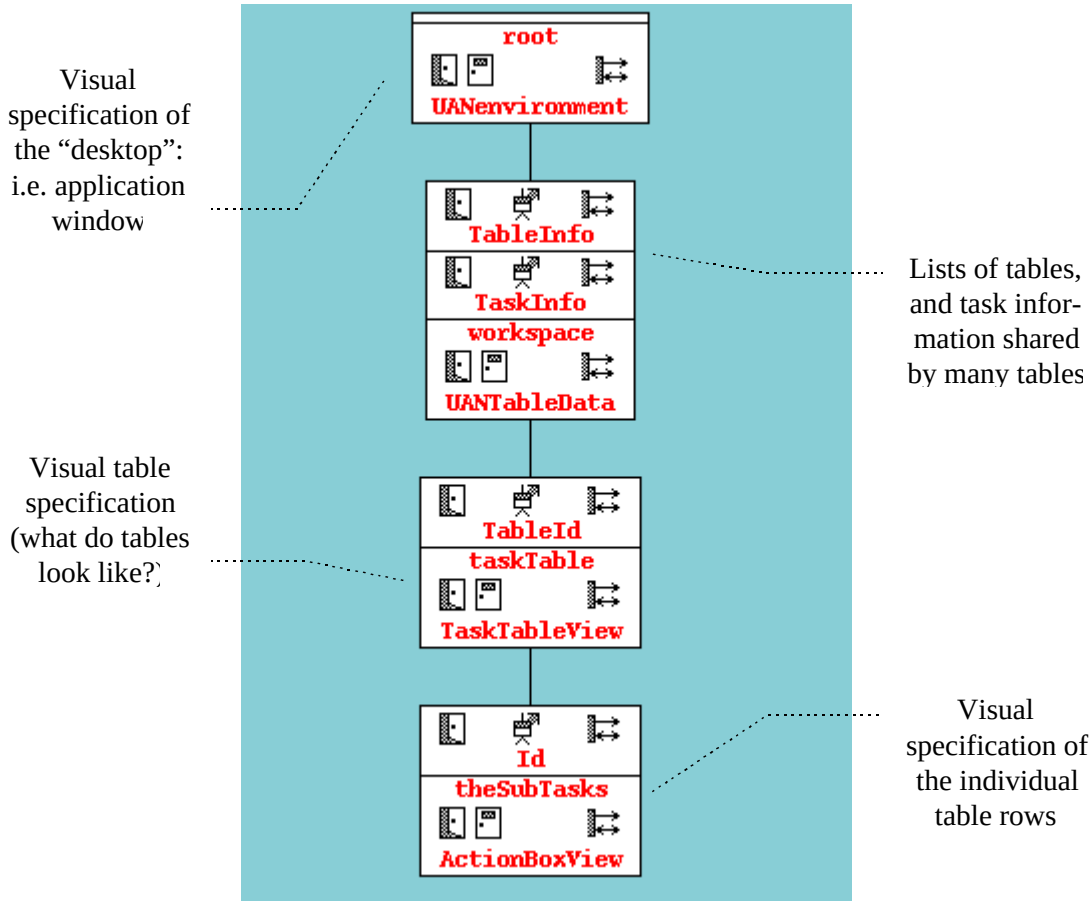

Figure 5: The UAN Browser during the first iteration.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TaskInfo: First Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Contains info pertaining to the list of tasks and their relationships.

%% We keep track of ActionLists.
type TaskData = [TaskId].
type State = [(TaskId, TaskData)].

allTaskNamesReq xs = map fst xs.

taskActionsReq xs n = ...

initially = [("Main Task",["Sub Task 1","Action 1",""]),
                        ("Sub Task 1",["Action 1","Action","Action 3"])].
```

Figure 6: The definition of TaskInfo in the first Iteration

The problem became apparent when I realized that I had to be able to repeat subtask identifiers. In the UAN, it is possible to have a terminal task (e.g. M^v) repeated within a single table. However, Clock does not allow two subviews of a view to have the same initialization string. This restriction would be violated if two sub-tasks of a TaskTableView had the same initialization string. The TaskInfo ADT had to be restructured.

The fact that supporting multiple table contexts required that a single table could be associated with many stacks (i.e., many different paths followed through ancestors in the graph) also became clearer, and led to another problem. The existing TableInfo ADT, also associated with the DeskTop component, was very similar to the TaskInfo ADT: a list of directory entries, with each entry consisting of a task identifier (the string of the task described by the table) as a key, and a list of task identifiers to denote the "stack" items as shown in figure 7. The problem, of course (clear in hindsight!) was that this scheme necessitated defining an entry in the list for each *context* in which a table appeared: potentially a lot of work. Further, if there were two or more contexts for any table in the specification, repeating the table's task identifier key in the list caused the duplicate sub-view problem to arise. Obviously, a serious re-think of the TaskInfo and TableInfo internals was required.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TableInfo: First Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Contains info pertaining to each task box on the desktop.

%% State is list of pairs: table name and task stack leading to table.
type State = [(TaskId, [TaskId])].

allTableNamesReq xs = map fst (debug xs).

tableTaskStackReq xs n = lookup n xs.

showTableUpdt xs (tableid, taskid) = ...

bringTableToFrontUpdt d tableid =
                        let p = lookupPos tableid d in
                                d#(1,p-1) ++ d#(p+1,length d) ++ [d@p]
                        end let.

initially = [("Main Task", [])].
```

Figure 7: The definition of TableInfo in the first Iteration

4.1.1.2 Summary

- Most of the difficulties I experienced in this iteration were caused by my unfamiliarity with ClockWorks.
- Despite this learning curve, I was able to rapidly specify the large-scale components of the application, based on a general idea of how I wanted the application to look.
- By the end of this iteration, the application allowed me to navigate multiple UAN tables by clicking on non-terminal subtasks, which would display the associated table.
- This iteration was halted when I realized that there were serious deficiencies in the internal data structures I had designed to keep track of task and table information.

The first iteration lasted from January well into February: the entire month of January was spent in administration, learning and practising with the declarative programming style, and getting ClockWorks to execute anything I wrote at all. Almost all of the above mentioned work was performed in the first and second week of February.

4.1.2 Second Iteration

4.1.2.1 The Details

During the rethinking at the start of this iteration, it became apparent that it would be sensible to have a "task library", with each task in the system identified by a unique number. This would simplify passing task information around (rather than concatenating strings, e.g. in the case of terminal tasks with multi-column text), and would also allow the user to access a task directly through a listing of the library contents. Similarly, it seemed appropriate to designate tables with a unique number as well, to avoid the multiple-context subview problem. I changed the TaskId and TableId types from strings to numbers, and introduced the RowId type, also a number. Figure 8 shows the changes made to the browser architecture in the second iteration.

To solve the duplicate subtask problem, I decided to change the TaskInfo state to a list of ordered pairs of the form ((TableId, RowId), TaskId), as shown in figure 9. The initial pair in the tuple would be guaranteed unique in the system, and since the row number would be guaranteed unique in any given table, state information could be passed to the ActionBoxView subviews as a unique number calculated by multiplying the RowId by some large constant, and adding the TaskId. The TaskId part of the combined number could then be extracted within the ActionBoxView.

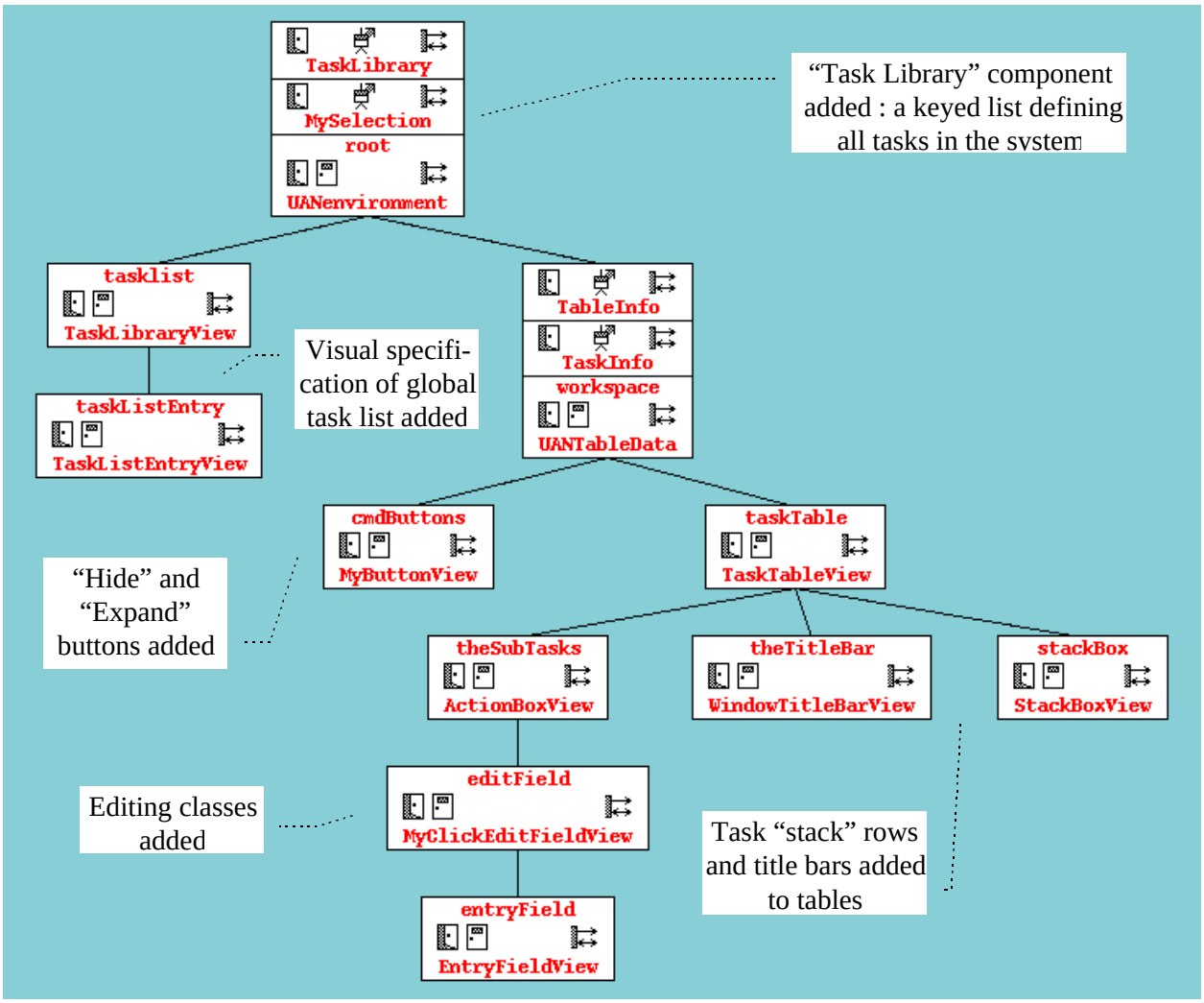

Figure 8: The UAN Browser during the second iteration.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TaskInfo: Second Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Tracks the task performed by each (table,row) pair in the system.

type State = [((TableId,RowId),TaskId)].

replaceTaskUpdt xs table row task =
        enterDict (table,row) task xs.

findTaskReq xs table row = ...

tableRowsReq xs table = findRows xs table.

findRows [] table = [].
findRows (x:xs) table = ...

%% given a TaskId, see if there is a table designated for the task (is it non-term?)
findTableReq xs task = findTab xs task.

findTab [] task = 0.
findTab (x:xs) task = ...

initially = [((1,titleRow),1),((1,titleRow+1),2),((2,titleRow),2),
        ((2,0),1),((2,titleRow+1),3),((1,titleRow+2),2)].
```

Figure 9: The definition of TaskInfo in the second iteration

Making this change altered the whole nature of the information flow between tasks and tables. Almost all of the information about both tasks and tables was now in TaskInfo, and TableInfo was reduced to a list of TableId/Boolean pairs, controlling whether or not the table was visible on the desktop (figure 10).

As is apparent from the code, TaskInfo was entirely rewritten, requiring a large number of changes to the updates and requests in the architecture. However, it was surprisingly quick: the redefinition of TaskInfo and TableInfo, and the introduction of the Task Library, took only two short days (approximately ten hours total) to put in place before the system was again working. The rest of this iteration was spent adding functionality to the system.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TableInfo: Second Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Contains info about each task box on the desktop, e.g. visibility

type State = [(TableId, Boolean)].

allTableIdsReq xs = map fst xs.

tableVisibleReq xs tableid = lookup tableid xs.

showTableUpdt xs tableid =...
hideTableUpdt xs tableid = ...
bringTableToFrontUpdt d tableid = ...
isFrontTableReq d tableid = ...

addTableUpdt xs tableid = xs ++ [(tableid,True)].

initially = [(1,True),(2,True)]
```

Figure 10: The definition of TableInfo in the second iteration

Table movement was implemented in this iteration through the addition of the WindowTitleBarView component (see figure 8). By this time, I had got the "hang" of ClockWorks, and table movement was put in place after one morning's work copying and altering code from a Critical Path Planner ClockWorks application developed by Graham [4]. Very little debugging was required. This was first time I became really impressed with how fast functionality could be added to ClockWorks. The same task in a C++/X environment would have taken far more work.

This iteration also saw the introduction of several "canned" classes provided by the ClockWorks team. I introduced a "Hide" and an "Expand" button (the MyButtonView component) to control the visibility of tables, and a "Replace" button to allow a selected table subtask to be replaced by a selected task from the library list. The operation of the buttons, as defined in the imported component, did not match what I desired in feedback characteristics, so I was forced to copy the code, rename the class, and change the visual function. Again, however, it was a matter of about an hour's work to alter the function of the button to my satisfaction.

What took longer was the incorporation of ButtonClick updates into the ActionBoxView and TaskTableView components. While trying to figure out how and when ButtonClick updates were to be accepted, it quickly became apparent that the system suddenly had a complex set of selection states: to hide a table, the table had to be selected; to replace a task, a subtask and a library item had to be selected; to expand a table, a subtask or a stack item had to be selected. Again, I found that the canned "Selection" component didn't provide the functionality I needed; although I initially tried to use the predefined component and have a separate selection request handler for tables, table rows, and task library list rows, I ran into a problem. The system contained an update in a component near the base of the architecture tree that needed the status of all three selections during processing, but was unable to direct the selection requests to the appropriate ancestors in the hierarchy. Eventually I decided to create a single "super" selection request handler (MySelection in figure 8), containing information on all three selection areas, and containing *nine* requests and updates, instead of the original three. Since this "super handler" was used by so many components in the system, it had to be placed at the top of the hierarchy. This was the point where I began to wish for rapid ways to add and remove requests and updates from the system.

Finally, this iteration also saw the introduction of editing. The introduction of the canned MyClickEditFieldView component was the most successful predefined component introduction encountered. I modified the ActionBoxView class into a horribly complex (in terms of the number of requests and updates) form, but the changes only took part of one day, since the scope of the changes was mostly local, and all of the hard interface work was already performed by the MyClickEditFieldView component. Once again, the speed of adding functionality was impressive; but the complexity of the ActionBoxView class became extremely high; for the architecture to be made 'clean' again would have meant creating a whole set of classes to divide up the functionality squashed into ActionBoxView.

This iteration ended with the realization that, while a lot of functionality had been added, the internal structure was inefficient and extremely complex, as indicated by the number of requests and updates in the system, and the awkwardness of some of the code, notably that in TaskInfo. While the duplicate subtask info problem had been solved, I realized that, although I could now represent any number of identical task tables with different stacks, for each different context I would have to store duplicates of the ((TableId,RowId),TaskId) pairs for all of the table's sub-task rows. Further, I still required the person keying in the specification to provide a table identifier for *every context* of every table.

The end of this iteration marked an interesting point in the development of the application. It was the most functional point in the application's life, but it also had the most confused internal structure. I decided at this point to focus on simplifying the internals, while providing a slightly restricted functionality by removing the editing and task-replacing functionality (tasks inappropriate for a browser).

4.1.2.2 Summary

- A "Task Library" was added, to allow keyed access to task names, and to provide a visible list of all of the tasks specified in the system.
- I decided to identify tasks and tables by unique numbers, rather than strings. Tasks were then associated with tables using a triple: ((TableId, RowId), TaskId). The table/row pair allowed me to uniquely identify repeated tasks in the same table, one of the problems identified in the first design.
- Table movement and table hide/expand buttons were quickly implemented using several "canned" classes provided by the ClockWorks team. Many of the predefined classes did not have behaviors matching what I wanted, so I had to rewrite parts of them. However, the rewriting was rapid.
- Editing of task names was also introduced. The predefined edit classes worked very well, with almost no modifications required.
- This iteration was halted when I realized that, although a lot of functionality had been added to the system, it was at the cost of architectural complexity. Many of my solutions to problems in the first iteration were proving awkward.

This iteration lasted from mid-February until mid-March, but only involved approximately a week (40 hours) of time spent at the computer. The speed with which I was able to make changes to the system drastically increased as my familiarity with ClockWorks grew.

4.1.3 Third Iteration

4.1.3.1 The Details

The third iteration was the most rapid. A morning's work at the chalkboard was dedicated to refining the internals. Once again, with a clearer understanding of the requirements, it was possible to drastically simplify the TableInfo and TaskInfo operations. The (table/row) task identifier method in TaskInfo was abandoned, and I returned to the initial TaskInfo state structure, as shown in figure 11. The duplicate subtask problem was addressed locally: when generating subviews, the table view was enhanced to generate sequential row numbers, and the unique combined row/task number was used as described above. The multiple-context problem was solved by maintaining a list in TableInfo (figure 12) of only currently visible tables, rather than trying to maintain a list of all possible tables.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TaskInfo: Third Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Tracks the tasks required to complete each non-terminal task in the
%% system.

%%-------------------------------------------------------

type State = [(TaskId,[TaskId])].

initially = [(1,[2,3]),(2,[4,5,6]),(3,[11,12,13]),(4,[10]),(6,[7,8,9])].

%%-------------------------------------------------------

allNonTerminalsReq xs = map fst xs.
isNonTerminalTaskReq xs taskid = (find (taskid, xs)) ~= 0.
taskAtRowReq xs taskid rowid = ...
subTaskListReq xs taskid = lookup taskid xs.

%%-------------------------------------------------------

replaceTaskUpdt xs taskid row newtaskid = ...
```

Figure 11: The definition of TaskInfo in the third iteration

In the current system, a unique table id is generated when the table is created. I tried using a predefined Counter ADT for this id generation, but settled instead for a function that simply returns the maximum number from a list of numbers plus 1. This latter approach keeps the operation totally local to the TableInfo request handler, and avoids adding another request handler to the architecture. Each TableInfo entry consists of this unique number and a list of task ids representing the stack. The last task id in the stack list is the task id of the table itself, and is therefore used to associate the table with the TaskInfo data.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TableInfo: Third Iteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Contains info pertaining to each table instance displayed on the desktop.
%% Entries are added and removed as table instances are created (shown)
%% and destroyed (hidden). Associated with each table id is a list of task ids,
%% representing the path of tasks followed to reach this table. The last task on
%% the list is the title task of this table.

%%-------------------------------------------------------

type State = [(TableId,[TaskId])].
initially = [(1,[1])].

%%-------------------------------------------------------

allTableIdsReq xs = map fst xs.
tableTaskReq xs tableid =...
tableStackReq xs tableid = ...
stackAtRowReq xs tableid rowid =...
isFrontTableReq d tableid = ...

%%-------------------------------------------------------

addTableUpdt xs tasklist = ...
removeTableUpdt xs tableid = ...
bringTableToFrontUpdt d tableid = ...
```

Figure 12: The definition of TableInfo in the third iteration

I also decided to focus on the "browser" aspect of the application, and removed the editing functionality, which was extremely dependent on the internal structures of the application defined in the second iteration. Adding editing, I realized, caused an explosion in the number of new tasks that the system would have to support. In the interests of clarity and simplicity, I decided to forego editing in the third iteration.

With this clearer view of the required functionality, it was extremely easy to remove extraneous components and clean up the architecture. As mentioned above, the user action information was associated with interface feedback, state, and computation calling information in the tables. This was done in the SubTaskBoxView component, which replaced ActionBoxView. The table title row was also given its own TitleBoxView component, since it differed visually from the other rows of the table. The resulting architecture is shown in figure 13.

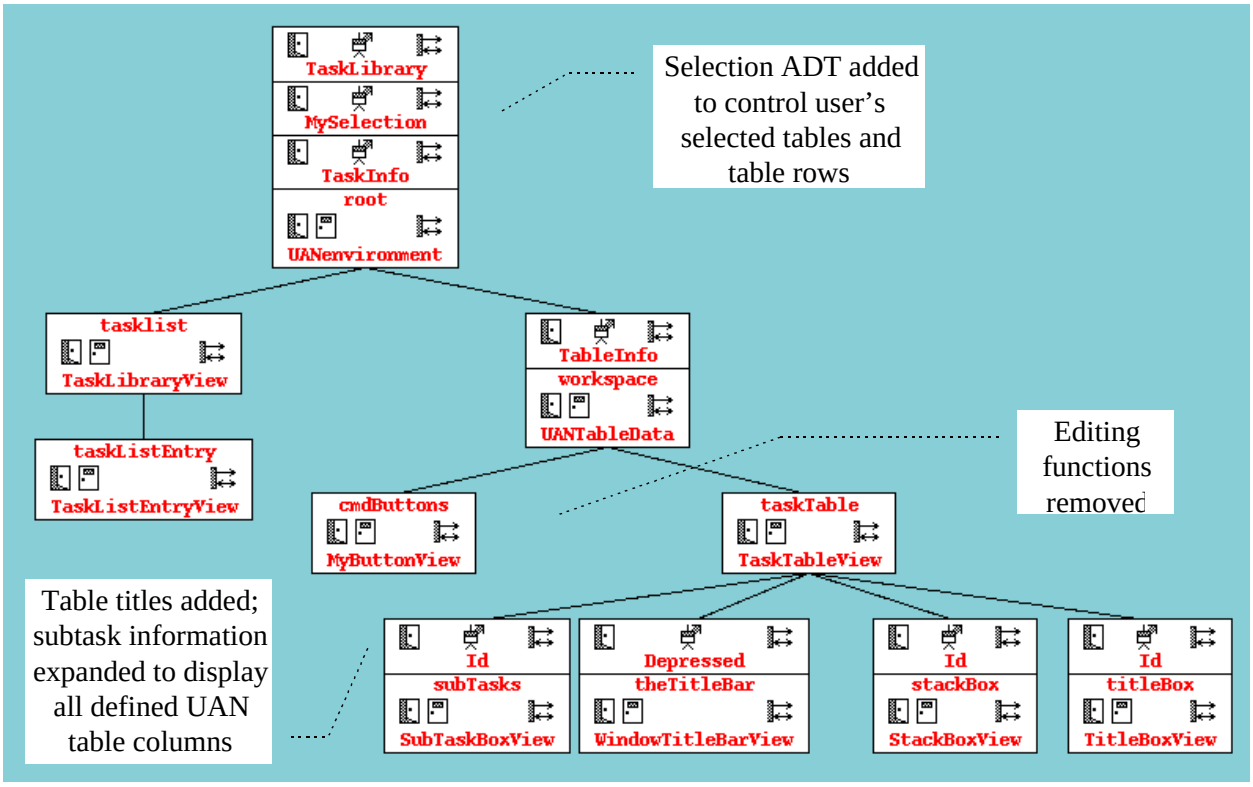

Figure 13: The UAN Browser during the third iteration

For the first time, significant effort was spent documenting and standardizing the style of the component text, and half a day was spent making alterations to the look and feel of the application (colours, text styles and sizes, etc.).

4.1.3.2 Summary

- This iteration was the most rapid. Some final rethinking of the relationship between tasks and tables was done, allowing me to drastically simplify the TaskInfo and TableInfo components.

- The editing functions, too complex to easily rewrite, were removed. Cleaning up and removing extraneous components was extremely quick and easy.
- The User Action information was finally associated with Interface Feedback, Interface State, and Connection to Computation information in a new SubTaskBoxView component.
- Significant time was finally devoted to standardizing the code style, and some time was spent tinkering with changes to the "look and feel (colours, fonts, etc.), of the application.

Overall, the third iteration took approximately two solid days of work.

4.2 The UAN Browser: current implementation and future work

The current browser software directly supports the task "View Table of Specific Task" identified in section 2.3. It provides a desktop upon which multiple tables can be concurrently displayed. Each table contains a complete specification of the table's title task according to the UAN standard. A specific task can be located by tracing subtasks in the visible tables: the user selects any non-terminal task in a visible table and presses the 'Expand' button. This makes the table of the selected non-terminal task visible. Alternately, the user can select the task by name directly from the task library list of all of the non-terminal tasks in the system, and pressing the 'New Root' button. Figure 14 shows a screen shot of the current application.

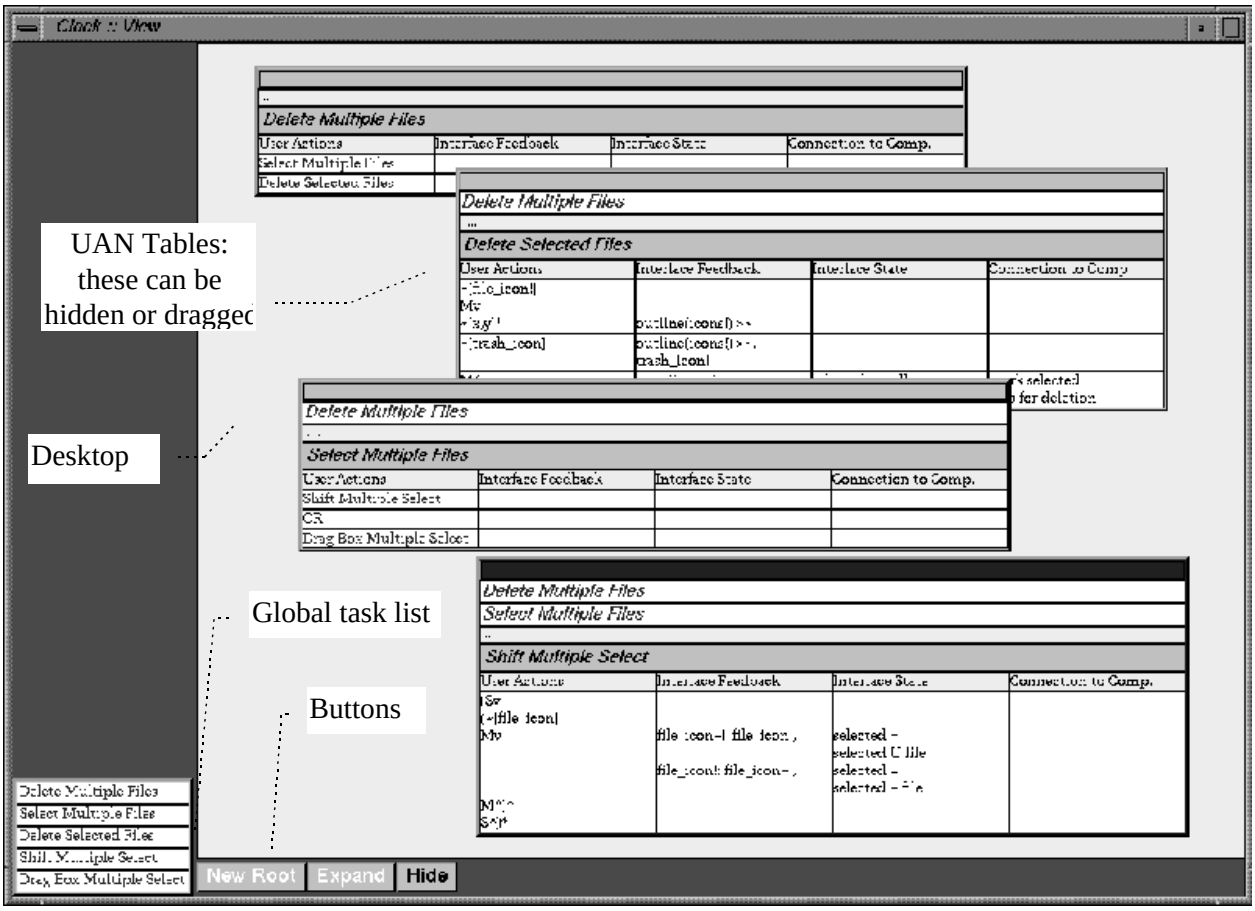

Figure 14: The user interface of the Browser

The "View Context of specific Task(s)" task is partly supported by the mechanism described above, when used in conjunction with the stack information presented with each table. The stack is the set of rows located above the table's title, each of which indicates an ancestor task selected in order to arrive at the current table. In essence, the stack shows the "path" (from most distant to most immediate ancestor, in order) followed in the graph to arrive at the current table, and therefore shows the context of the table's task. It is therefore possible (although still inconvenient) to identify all of the contexts of a single task: for example, the user can display the table for task D - found by traversing task A and task B - right beside another table for task D found by traversing tasks F and G. The two tables are identical, except for the stack information.

Each task on the stack is, of course, a non-terminal task, and so is also selectable and expandable by the user, allowing a two-way traversal of the task hierarchy. If the desktop gets too cluttered with tables, the user may hide any table by selecting it and pressing the 'Hide' button. Note that if a table is created with the 'New Root' button from the task library list, the system does not attempt to place the context of the table: it is considered to form the root of an extracted portion of the task hierarchy (hence the cryptic name). Tables can be moved around the screen by clicking and dragging on the title bar of the table window. Tables cannot currently be iconified.

The current system provides no support for the third major subtask, "View Section of Task Hierarchy". One solution to this inability to extract structural information would be to provide another work surface in addition to the desktop displaying a low-detail overview of the task hierarchy. The desktop could continue to provide information on table details, while the new space could provide the graph-specific information required. Alternatively, some functions could be tied to the tasks themselves: for example, the user could select a task from the task list and have the system display a table for each context in which the task occurs in the system.

I believe that the current system constitutes an acceptable start towards a fully-featured tool. However, there is obviously a lot left to be done to implement the full usefulness of a graph-structured UAN browser/editor.

5 ClockWorks: pros and cons for the Lazy Programmer

This section contains a series of observations about how the ClockWorks approach aids and/or irritates the lazy programmer.

5.1 The Speed vs. Structure Debate

Tools such as ClockWorks that enforce a specific architectural style tend to produce clearly-defined and well-understood applications. This contrasts favourably with tools such as Microsoft's Visual Basic [8] that favour speed over structure: these tend to produce applications that are architecturally singular (any two applications will have dissimilar architectures). This type of application is hard to understand, since to understand the application structure, anyone other than the original programmer probably has to start from scratch.

The ClockWorks tool also allows for smoother software evolution by forcing the system structure to keep pace with the addition of function to the application over time. ClockWorks makes it difficult to "fudge" functionality on a bad internal system structure. Development reaches a point (in my case at the end of the second iteration) where the messiness and inefficiency of the system is totally, *visually* apparent, and the amount of work required to add new functionality to the mess begins to outweigh the amount of work required to streamline the structure. The end result is that the application's functionality remains based on clearly defined and understood internals. This kind of functionality is easy to adapt and extend. The evolutionary 'smoothness' of ClockWorks applications contrasts favourably with the Visual Basic group, whose applications' system structures often freeze in an early and incomplete form as it becomes too difficult to reorganize the growing web of workarounds into a new, coherent structure. Over time, the functionality of such applications becomes increasingly based on workarounds, and changes or extensions become a game of side-effect Russian roulette.

However, enforcing a specific architecture has drawbacks. By restricting the use of workarounds and short cuts, ClockWorks sometimes is a frustrating tool for the lazy programmer. Having to stop and rethink fundamentals, and possibly face the fact that a good portion of the system function that has been so painstakingly developed must be scrapped, is not something a lazy programmer wants to do. This builds resentment against the tool, and may, especially with an inexperienced ClockWorks user, give rise to the belief that the tool is limited in power, since the programmer's perception then becomes "there's no easy way to do what I want to do".

The drawback just mentioned concerns programmer perceptions. There is another, more serious, related drawback. If a tool strictly enforces an architecture, then the tool does in fact (not just in programmer perception) become limited by any weaknesses of that architecture. The architecture designer has placed himself in control, and has developed a set of rules about how systems can be structured. The usability of the whole tool becomes tied to the completeness and power of that set of rules, since the programmer can't develop extra-architectural solutions to problems not manageable within the architecture. ClockWorks enforces an extremely well thought-out architecture: care in architecture design is the only way to minimize this problem.

Strict enforcement of an architecture causes programmer claustrophobia. The biggest attraction of the free-form tools such as Visual Basic is the belief that a programmer using them can always solve a problem his own way, and that there are always several optional solutions available. With restricted architectures there is a greater risk that, even if there is a way to solve a given problem, a less experienced developer may be unable to find it.

I encountered these types of drawbacks when dealing with the Selection ADT, as described in the "Second Iteration" section, above. Here was a requirement that I felt was legitimate: I wanted to be able to make three requests within an update handler, to obtain the selection status of three elements of the system. There was a predefined Selection ADT that initially seemed to fit the bill, and a logical place in the hierarchy for each of the three required instances of Selection. However, I found that since the architecture automatically routed my request messages (very convenient 99% of the time), I couldn't control the delivery of the requests to a specific handler. Not specifying a specific message recipient does prevent tight coupling between components and facilitates changing the program structure, but not having the option to specify a recipient forced me to produce either one enormous new ADT customized to the application, or three customized variations of Selection -- a lot more work than managing a specific address in the update handler.

If I have a suggestion about fundamentals in ClockWorks, it is this: I think that it would be worthwhile to take a second look at the strongly restrictive policies, specifically message addressing and scoping (i.e. which components can see a request/update handler), with a view to relaxing the architectural requirements as much as possible without compromising ClockWorks fundamentals. This would increase the variety of solutions available to programmers: it probably wouldn't result in much additional real functionality, but programmers like to have multiple options when solving problems. If some of these restrictions are *carefully* relaxed, I think that additional real (and perceived) flexibility and power can be added to ClockWorks. For example, the hiding mechanism that has been proposed for request handlers within groups would be very useful if implemented.

Personally, I believe that the tradeoff of short-term development speed for the ability to maintain a clear internal system structure is worthwhile, and in fact repaid over time. I also believe that this tradeoff will not be acceptable to the lazy programmers of the world outside the pure research community if the tool is viewed as rigidly enforcing a specific architecture.

5.2 The Visuals

I found the DisplayView type to be a surprisingly powerful abstraction. I didn't run into anything that I couldn't figure out how to display (except a minor X pane problem with my brief foray into menu use at the end of the project, solvable but for lack of time). I still find the left-to-right order of evaluation in DisplayViews confusing, given that order of evaluation in the declarative paradigm works the other way. If this order of evaluation were reversed (I realize that this would make incremental updates harder or impossible: this is just hypothetical), it would also allow class subtypes to redefine some or all of the view definition of the supertype; whether this would be useful or just dangerous is not clear.

I found that my ability to reuse predefined visual components was relatively low. In almost every case, I ended up making a local copy of the predefined component and making changes to the component function. ADTs were more portable than the visual classes, but even these sometimes couldn't be made to work. Part of the problem was due to my inexperience, and part due to the complexity of the class inclusion process. It often wasn't clear just which classes of an apparently associated set of classes would be required to provide the desired function, especially when borrowing classes from other projects that used grouping. Even once the classes were included in my project, there inevitably seemed to be extraneous or missing requests/updates or handlers: the predefined classes did too much, or not enough.

When dealing with visually-oriented components like buttons, of course, class re-use can be expected to drop. These classes are more like widgets, and common wisdom promotes customizing widgets into specific sets rather than placing a vast amount of generalized function into them and making them too cumbersome to use. In the case of ADTs, however, re-use is a worthwhile goal.

My suggestion on the topic of reuse is simply to continue the evolution of a component library, and to streamline the inclusion process: make it easy for a programmer to try the predefined components.

The ClockWorks visual development environment was *very* useful, if not essential.

- It facilitated navigating through the system. During coding, it provided a means of single-click access to any code module in the system. My only navigational suggestion would be to provide editor-specific (e.g. emacs) extensions to ClockWorks. In the case of emacs, it would be nice to operate within a single

session (it is much faster to create/destroy buffers or frames than entire editing sessions), and it would be useful to have a Clock syntax checking module similar to the C++/Lisp modules available.

- It allowed efficient complexity management by allowing the user to display or hide the visual representation of the components and messages.

- It served as a memory aid, both during a development session between sessions. Even after a week away I was able to sit down and recall the last state of the project's structure within a few minutes.

- Interestingly, it served to shame me when the structure was in a bad state. Many symptoms of poor Clock structure (overly complex components with enormous request/update lists, poorly named components, etc.) are embarrassingly visible. By the end of the second development iteration, I couldn't stand to look at the mess anymore--the visual interface goaded me into cleaning up the structure. This raises the question of just how much sloppiness in development would vanish if programmers had to display their application's structure and code in front of a critical audience every day.

My suggestions for the visual aspects of ClockWorks are straightforward, and have probably already been identified.

- Make the type information of updates/requests and ADTs easily accessible, e.g. via one or two button clicks. This type information is hard to remember, and I found myself constantly going through the complex task of calling up a request or update definition just to remember the type of its parameters.

- More thought needs to be given to the definition, addition, and removal of requests and updates. Fewer menu selections should control this process - I was constantly getting into the update list when I wanted the request list, etc. Perhaps one dialog allowing access to both updates and requests would be more efficient. Direct manipulation of request and update graphical objects would be ideal: the user could select a displayed update object, copy it around, move it, delete it. etc. Also nice would be some way to identify and manipulate sets of requests/ updates: for example, it would be nice to be able to delete all instances of a given request/update from the system in one operation, or to identify all components using a specific request/update.

5.3 Miscellanea

I found debugging to be very fast: Clockworks provides minimal tool support for debugging, but what is provided is adequate, since there are a minimal number of *types* of bugs that are encountered due to the architectural restrictions. The number of times major changes ran the first time was encouraging. All the debugging tools have to do is let the programmer know where the problem is located, which they do admirably. This is not to say that all of the work is done: there are still type errors that aren't caught until runtime, such as the X-degrading 'above' function problem mentioned previously. I would strongly support the inclusion of type-checking.

There are some problems in the 'above' and 'aboveStretching' functions when one of the DisplayViews in the list evaluates to a NoView. I have replaced the aboveStretching code: the version that works with NoViews is in the Globals file for the project.

ClockWorks is very easy to learn, if harder to master. Less experience in applying "technique" is required than in a low-level imperative language environment - since the specification language is high-level, there are a restricted number of ways to accomplish each programming task. This results in a rapid learning curve: I began to feel comfortable with the concepts quickly, and was able to improve the quality of (i.e. simplify) my implementation considerably in each iteration. In Clock, the best way to do things usually involves basic concepts, unlike many imperative languages, where efficiency is often tied to complex concepts and cryptic coding techniques.

6 Conclusion

ClockWorks embodies a very well thought out approach to the problems of incremental interactive software development. The approach produces software that is structurally precise and therefore all the things software should be: extendible, maintainable, understandable, reusable. ClockWorks's approach is also risky. Many of the decisions that went into its design, such as the use of a declarative language and the enforcement of a specific architecture, cut both ways: they form the basis of the quality of the current product, but they reduce the audience to which the product appeals. I, however, am among that audience--I enjoyed working with ClockWorks very much.

With regard to the UAN, it seems apparent that the amount of value provided by UAN specifications in the development process is tied directly to their ability to evolve concurrently with the system specification. This ability to evolve is not inherent in the notation; UAN requires sophisticated tool support if it is to realize its potential. The belief that UAN can be effectively used with basic text-processing tools is wishful thinking: too much of a specification's information is unavailable or obscured in such a form.

The incorporation of UAN tool support into the ClockWorks environment appears to be a worthwhile goal; supporting both task-oriented and behavioural specification in one environment provides the lazy programmer with two convenient and powerful ways of looking at development problems.

7 References

[1] H. Rex Hartson, Antonio C. Siochi, Deborah Hix. *The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs*. ACM Transactions on Information Systems, 8(3):181-209, July 1990.

[2] T.C.N. Graham, C.A. Morton, T. Urnes. *ClockWorks: Visual Programming of Component-Based Software Architectures*. Journal of Visual Languages and Computing, Academic Press, July 1996 (to appear).

[3] T.C.N. Graham. *The Clock Language*. Reference Manual. Electronic Technical Report CS-ETR-95-01, Department of Computer Science, York University, June 1995.

[4] T.C.N. Graham et. al. *The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation*. Department of Computer Science, York University (in progress).

[5] T.C.N. Graham, T. Urnes. *Linguistic Support for the Evolutionary Design of Software Architectures*. Technical Report In Proceedings of the Eighteenth International Conference on Software Engineering. IEEE Computer Society Press, Berlin, Germany, pp. 418-427, March 1996.

[6] P. Hudak, J. Fasel. *A Gentle Introduction to Haskell*. Department of Computer Science, Yale University, 1992

[7] Borland International Inc. *Borland C++ User's Guide*. Borland International, 1993.

[8] Microsoft Corporation. *Visual Basic User's Guide*. Microsoft Press, 1996.

Appendix A: A UAN Specification for the implemented UAN Browsing Tool

TASK: Browse UAN Specification			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
({ View Table of Specific Task }			
⇔			
{ View Context(s) of Specific Task(s) in Task Hierarchy }			
⇔			
{ View Section of Task Hierarchy }	**NOT IMPLEMENTED**		
)+			

TASK: View Table of Specific Task			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
(Rearrange Desktop			
{ (Select Task via Table on Desktop			
OR			
Select Task via Task Library List)			
Display Table })*			

TASK: View Context(s) of Specific Task(s) in Task Hierarchy			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
(Rearrange Desktop			
{ (Select Task via Table on Desktop			
OR			
Select Task via Task Library List)			
Display Table })*			

TASK: Rearrange Desktop			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
(Bring Table To Front			
OR			
Move Table			
OR			
Hide Table) *			

TASK: Select Task via Table on Desktop			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
(~[table.stack_ item]			
Mv^)	displayOnTop(table) table.stack_item-! : table.stack_item! ∀ table.rows'! : table.rows'-! expand_button-! : expand_button!	sel_table_task = table.stack_item sel_table = table frontOfList(table) in TableInfo list	
OR			
(~[table.non-term_sub_task]			
Mv^)	displayOnTop(table) table.non-term_sub_task-! : table.non-term_sub_task! ∀ table.rows'! : table.rows'-! expand_button-! : expand_button!	sel_table_task = table.non-term_sub_task sel_table = table frontOfList(table) in TableInfo list	

TASK: Select Task via Task Library List			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
~[library list.row]			
Mv^	library_list.row-! : library_list.row! ∀ library_list.row'! : library_list.row'-! new_root_button-! : new_root_button!	sel_lib_task = library_list.row	

TASK: DisplayTable			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
(~ [expand_button]			
Mv	expand_button! : expand_button!! table_sel_task <> nulltask : displayOnTop(sel_table_task)	addTable(sel_table_task, sel_table_stack) to TableInfo list	
M^)	expand_button-!!		
OR			
(~[new_root_button]			
Mv	new_root_button! : new_root_button!! sel_lib_task <> nulltask : displayOnTop(sel_lib_task)	addTable(sel_lib_task, NULL) to TableInfo list	
M^)	new_root_button-!!		

TASK: Bring Table To Front			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
~[table]			
Mv^	table-! : table! ∀ table'! : table'-! displayOnTop(table) hide_button-! : hide_button!	sel_table = table, frontOfList(table) in TableInfo list	

TASK: Move Table			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
~[table.title_bar]			
Mv	table-! : table! table.title_bar!! ∀ table'! : table'-! displayOnTop(table) hide_button-! : hide_button!	sel_table = table, frontOfList(table) in TableInfo list	
~[x,y]*	table > ~	tablePos(table, x,y)	
M^	table.title_bar-!!		

TASK: Hide Table			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
Bring Table To Front			
~[hide_button]			
Mv	hide_button! : hide_button!! erase(sel_table)	remove_table(sel_table) from TableInfo list sel_table = nullTable sel_table_task = nulltask	
M^	hide_button-!! hide_button-! expand_button! : expand_button-!		