The *Vista* Environment for the Coevolutionary Design of User Interfaces

Judy Brown
Victoria University of Wellington
P.O. Box 600
Wellington, New Zealand
+64 4 471 5328
Judy.Brown@mcs.vuw.ac.nz

T.C. Nicholas Graham, Timothy Wright
Dep't of Computing and Information Science
Queen's University
Kingston, Ontario, Canada
+1 613 545 6526
{graham,wright}@qucis.queensu.ca

ABSTRACT
User centered design requires the creation of numerous design artifacts such as task hierarchy, task-oriented specification, user interface design, architecture design and code. It is increasingly accepted that such artifacts cannot be created in isolation, but instead incrementally coevolve, where information obtained from the development of one artifact contributes to the development of the others. In user interface development, these artifacts are typically developed by different people with different backgrounds, hindering the communication necessary for coevolution. This paper demonstrates how different design artifacts can be linked, exposing their common elements. Such links can be developed despite the differing points of view and differing levels of detail of the design artifacts. This paper describes *Vista*, a prototype tool for examining the links between design artifacts, and demonstrates how making these links explicit supports coevolutionary design.

Keywords
Task-analysis, task-oriented specification, user-interface, architecture, methodology

INTRODUCTION
User-centered design consists of a set of steps such as task analysis, user interface design, task-oriented specification, and implementation. Each step results in a design document that expresses information contributing to the usability of the resulting system. In the software engineering community, it is widely acknowledged that these steps cannot realistically be performed in strict sequence, but are instead carried out concurrently [2,7,13,16]. In the development of user interface software, the requirements of iterative refinement make the need for concurrent development even more pronounced.

If there is continuous feedback between the evolving design representations, each design activity can benefit from information derived in the other steps. For example, user interface design benefits from task analysis; problems in the task analysis can in turn be revealed during user interface design, allowing benefit to be derived in both directions. Fischer et al. [7] refer to concurrent design with multidirectional information flow as the *coevolution* of a set of design artifacts.

Figure 1 shows part of a process for the coevolutionary design of interactive systems. HCI designers and software engineers concurrently produce a set of design artifacts contributing to the system's implementation. These artifacts include task hierarchy, user interface design, task-oriented specification (in the User Action Notation [11]), software architecture and code. A complete process would involve the user community, for example through usability testing.

In our experience, the communication central to coevolutionary design is hindered by the difficulties of relating the design artifacts. Designers cannot see how knowledge from one design artifact can be applied to other artifacts if they cannot easily correlate the information provided in the two artifacts. Designers have difficulty correlating design artifacts since they express information from different points of view (e.g., the users' view vs. the implementer's view) and at different levels of abstraction (e.g., code is much more detailed than architecture) [8].

This paper shows how different user interface design artifacts can be linked to expose their common elements, facilitating the communication on which coevolutionary design is based. To support these links, we have developed *Vista*, a tool that allows designers to simultaneously navigate hyperlinked task hierarchy, task-oriented specification, software architecture and code documents. In Vista, links are developed partly by hand and partly through automated analysis of the design artifacts. The paper is organized as follows. We first introduce an application to be used as an example throughout the paper. We then introduce the design representations supported by the method. The following section explains how these artifacts are linked in *Vista*. The final section analyses the effectiveness of linking design artifacts in supporting coevolutionary design of user interfaces.

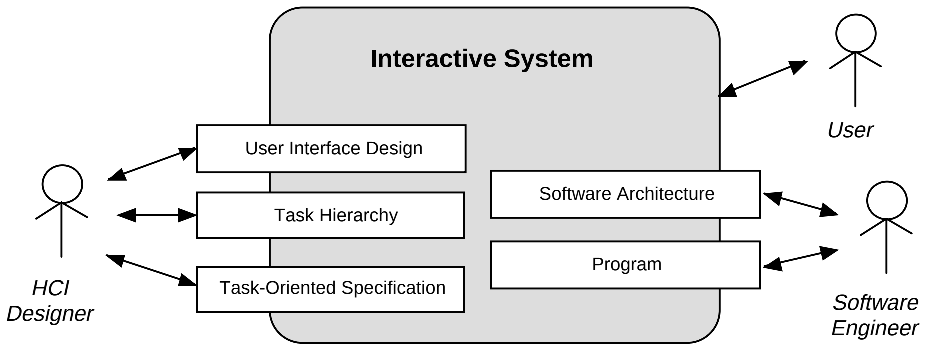


Figure 1: Coevolutionary design of interactive systems. The design process leads to a set of artifacts, which are developed iteratively and incrementally. Improvements in one artifact contribute to improvements in the others. Different artifacts represent the point of view of different participants in the design process.

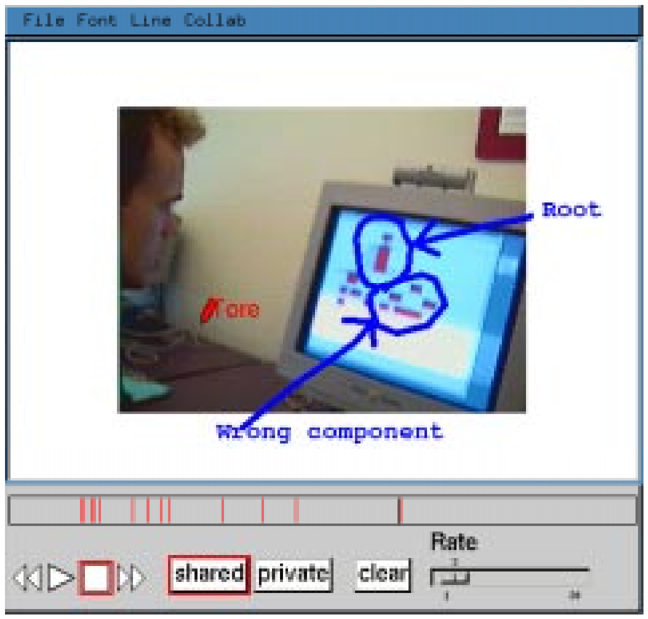


Figure 2: A collaborative tool for annotating videos.

VIDEO ANNOTATOR

Throughout the paper, we use the example of a groupware video annotation tool [9] to illustrate the use of coevolutionary design. The video annotator allows a group of people to review a video that has been previously recorded and add textual and graphical annotations to frames of interest. All participants in the session see the video playing, and see the annotations of other participants as they are added in real time. Standard VCR controls allow the video to be stopped, fast-forwarded and reversed. A *gestalt* view shows a timeline of the video, in which vertical lines indicate where frames have been annotated. When a participant clicks on one of the vertical lines, the view of all participants moves to the corresponding annotated frame. The annotator was developed to support the collaborative analysis of taped usability testing sessions.

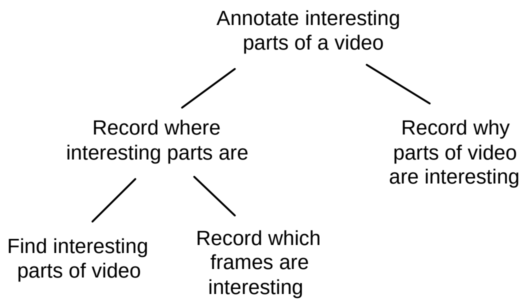

Annotate interesting parts of a video
Record where interesting parts are
Record why parts of video are interesting
Find interesting parts of video
Record which frames are interesting

Figure 3: An excerpt from the hierarchical task analysis developed for the video annotator of figure 2.

DESIGN ARTIFACTS

In order to demonstrate how different design notations can be linked, we first review the notations supported by our *Vista* tool and used in the process of figure 1. We concentrate on the three notations of task hierarchy, task-oriented specification and user interface architecture. We chose these notations for the initial version of Vista because they are well defined and because we believe that tool support for these notations could help them become more widely used. We hope to extend Vista in the future to handle other forms of design artifact, such as screen designs, scenarios and videos of usability testing sessions.

Task Analysis

Task analysis is widely used as a means of codifying the goals and tasks of the intended user community [1, 5, 14]. A design artifact resulting from task analysis is a hierarchical task decomposition.

Figure 3 shows an abbreviated task hierarchy for developing an annotated video transcript. The main task is to add annotations to interesting parts of the video. A task hierarchy such as this one shows how the main goal of the users can be decomposed into subtasks that help to achieve this goal. For example, in figure 3, the main goal is decomposed into the subtasks of noting where the

TASK Annotate interesting parts of a video
USER ACTIONS
Record where interesting parts are
|| Record why parts of video are interesting

TASK Jump to annotation		
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE
~[annotation_mark] Mv	Display current frame and current annotations	Current frame := frame for annotation mark
M^		

Figure 4:UAN tables describing how two tasks are carried out using the video annotator of figure 2.

interesting parts of the video are, and recording *why* these parts are interesting.

Task-Oriented Specification

A task-oriented specification is based on a hierarchical analysis. The task-oriented specification encodes how these tasks are carried out with the user interface, from the point of view of a user of the system. In the User Action Notation (UAN) [11], tasks are encoded in a tabular format. Figure 4 shows how the main goal of the system is encoded as a UAN task: in order to annotate interesting parts of the video, the user records where the interesting parts are, and records why those parts are interesting. These are specified under the heading of *User Actions*, as they represent the actions that the user performs in order to carry out the task. The two subtasks are connected by the "||" symbol, indicating that they may be carried out in any order, or even concurrently.

Task-oriented specifications explain how a user carries out these tasks with the user interface. For example, part of navigating the video involves jumping to a frame that has already been annotated, perhaps to add new annotations. This task is shown in figure 4. The user moves over one of the annotation marks in the gestalt view of the video (~[annotation_mark]) and depresses the mouse button (Mv). The video annotator jumps to the selected frame and shows the correct annotations. The *Interface Feedback* column specifies how the system responds to the user's actions; the *Interface State* column records (in natural language or pseudo-code) the effects of the user's actions on the internal state of the user interface. Finally, the user releases the mouse button (M^), completing the task.

UAN specifications provide precise descriptions of the dynamic aspects of user interfaces. They show how a system is used by walking through the tasks that users wish to perform, showing precisely what users may do, and

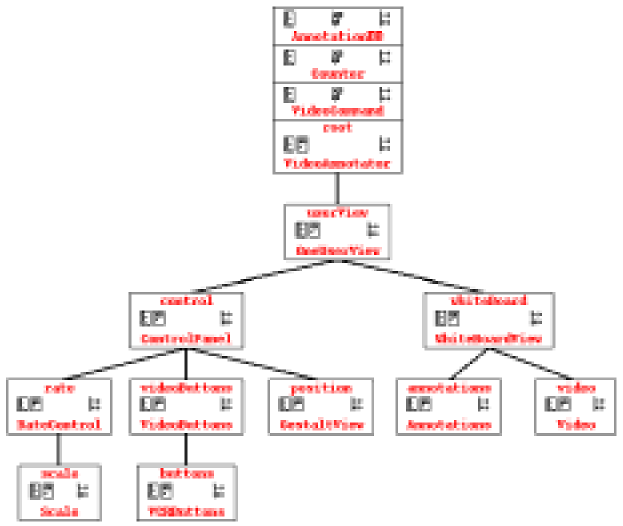


Figure 5: An excerpt from the software architecture of the video annotator of figure 2.

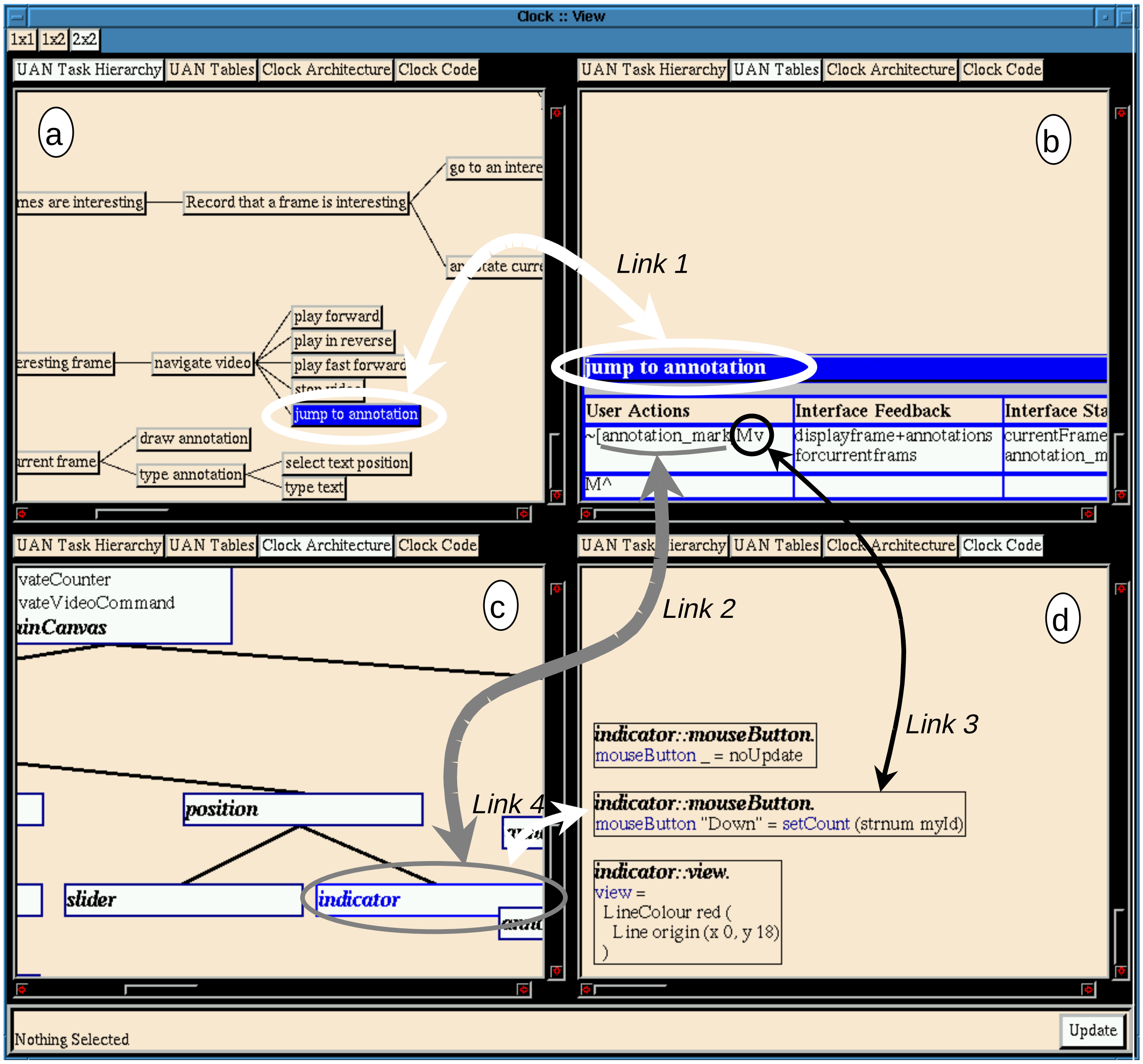
showing how the user interface responds to the user's actions. The UAN is a useful tool to HCI designers, as it allows the behavior of a user interface to be documented prior to implementation. UAN specifications provide a basis for evaluating user interface designs using cognitive walkthrough.

User Interface Architecture

Software architectures provide a means of decomposing complex systems into smaller parts, simplifying implementation, evolution and maintenance. A number of software architecture styles have been proposed to help in the development of interactive software, particularly the development of multiuser and multimedia applications [3,9, 12,17]. By using an existing architecture style, software engineers benefit from high-level structuring techniques developed by others.

In *Vista*, we use the Clock architecture style for developing user interfaces [9]. This architecture style provides high-level support for distribution, networking, concurrency control and multimedia, freeing developers from low-level programming details. As with a number of other modern architecture styles for user interfaces, Clock architectures are organized as layered model-view-controller (MVC) [15] clusters.

Figure 5 shows an excerpt from the Clock architecture implementing the video annotator. The architecture structure is compositional: the video annotator is composed of one *userView* per user; the user view is composed of a control panel (*control*) and a white board (*whiteBoard*). The architecture is composed of components, which are responsible for processing user inputs and maintaining the display. For example, *video* displays the current video frame, while *annotations* displays any annotations that have been made on that frame. Components may be augmented with *ADTs* which contain application data. For example, the



User Actions	Interface Feedback	Interface Sta
~[annotation_mark] Mv	displayframe+annotations forcurrentframs	currentFrame annotation_m
M^		

```
indicator::mouseButton.
mouseButton _ = noUpdate

indicator::mouseButton.
mouseButton "Down" = setCount (strnum myId)

indicator::view.
view =
  LineColour red (
    Line origin (x 0, y 18)
  )
```

Figure 6: Linked views in Vista. A designer has selected four views of the video annotator application: (a) the task hierarchy; (b) the task-oriented specification in the UAN; (c) the Clock architecture of the application, and (d) the Clock code. The Vista display has been annotated to show some of the links between the design representations. This display has been hand-annotated to show some of the links available to designers and developers.

root component is augmented with the *VideoCommand*, *Counter* and *AnnotationsDB* ADTs. The *AnnotationDB* ADT records which frames are annotated and how, while the *Counter* ADT represents the frame counter.

Components communicate via messages. For example, the *position* component responds to mouse button input (when a user clicks an annotation mark to move to a previously annotated frame.) *Position* uses methods defined in AnnotationDB and Counter to update the display.

Task-oriented and architectural specifications capture similar information. They both specify the functionality of the user interface, but from different points of view. The next section discusses the links between UAN task-oriented specifications and Clock architectures.

LINKS BETWEEN DESIGN REPRESENTATIONS

Figure 6 shows four views of the video annotator as displayed within the *Vista* environment. These views show the design and implementation of the *gestalt* timeline of the video annotator (figure 7). The *Vista* display is split into

Figure 7: The gestalt view of the video annotator (excerpted from figure 2).

four panels showing (a) the hierarchical task-analysis, (b) the UAN task-oriented specification, (c) the Clock architecture and (d) the Clock code of the gestalt view.

The gestalt view supports the task of navigating the video, shown in the task hierarchy panel (*a*). The UAN table of panel *b* shows that users perform this task by selecting an annotation mark along the timeline. The timeline is implemented by the *position* architecture component (shown in panel *c*). The detailed Clock code for this component is shown in panel *d*.

The views of these four panels are connected by hyperlinks, so that when designers click on information in one panel, the corresponding information is shown in the other panels. Figure 6 has been annotated to show four of the hyperlinks available to designers. For example, clicking on the "jump to annotation" task in panel *a* pops up the corresponding UAN table in panel *b* (link 1 in figure 6).

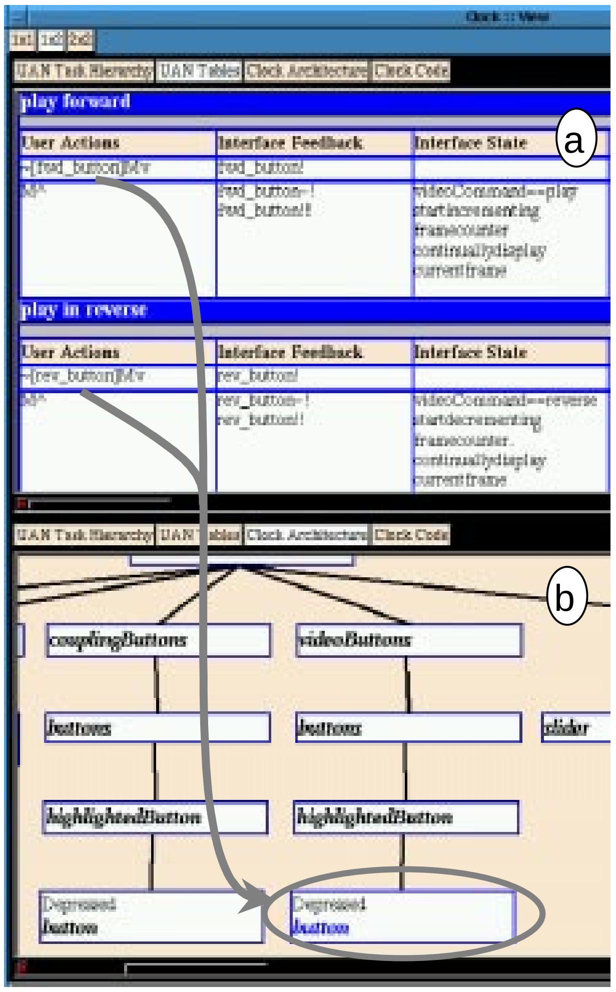
Designers may tile the *Vista* display into one, two or four panels, allowing them to select between more views and bigger panels.

The next section describes the links between the user-centered UAN task-oriented specification and the system-centered Clock architecture specification. Links between other specifications are not as complex. The section explores how the UAN concepts of *context*, *user action*, *interface feedback* and *interface state* are linked to the architectural concepts of *components*, *input events*, *view methods* and *ADTs*.

Linking UAN Contexts to Architecture Components

Clock architecture components implement UAN contexts. For example, in the task "Jump to annotation" of figure 6, the UAN context *annotation_mark* is implemented by the *indicator* architecture component. In *Vista*, clicking on *annotation_mark* in the UAN table highlights the *indicator* component in the architecture view.

Identifying which architecture component implements each UAN context is non-trivial. The HCI designer and the software engineer may have given the same entity a different name (as in the example above). Another problem is that UAN contexts and architecture components do not match on a one-to-one basis. Architecture designers do not necessarily create a new component whenever the UAN specification identifies a new context. Figure 8 shows how one architecture component can implement more than one UAN context. The *button* component of the video annotator supports the *play forward* and *play in reverse* tasks by implementing the *fwd_button* and *rev_button* contexts identified in the UAN task oriented specification. *Vista*



play forward

User Actions	Interface Feedback	Interface State
~[fwd_button]Mv	fwd_button!	
M^	fwd_button-! fwd_button!!	videoCommand==play startincrementing framecounter continuallydisplay currentframe

play in reverse

User Actions	Interface Feedback	Interface State
~[rev_button]Mv	rev_button!	
M^	rev_button-! rev_button!!	videoCommand==reverse startdecrementing framecounter continuallydisplay currentframe

Figure 8: UAN contexts fwd_button and rev_button are both implemented by the button architecture component.

solves the two problems described above by statically binding contexts to components.

Linking UAN Actions to Architecture Components

In a UAN dialogue specification, user actions specify what inputs users perform in order to carry out their task. User actions occur within contexts. For example, in order to jump to a previously annotated frame in the video, the user positions their pointer over an annotation mark (~[annotation_mark]) in the gestalt view and then depresses the mouse button (Mv). The "Mv" action therefore occurs in the context of the annotation mark. User actions are processed by the architecture component implementing the action context. The *indicator* component implements the annotation mark context, and therefore processes the "Mv" user action. Figure 6(d) gives the code of the *mouseButton* method contained in the *indicator* component. The method responds to a mouse button "down" action by issuing a *setCount* message, which updates the current frame number.

User actions are therefore linked to the components that implement those actions. Clicking on the "Mv" symbol in figure 6(b) displays the method that handles mouse button events (link 3), and highlights the component that contains the method (link 4).

Linking UAN Interface Feedback Specifications to Component Views

The column in a UAN table labeled "Interface Feedback" specifies how the display should change in response to user input. The software engineer uses this information to implement the view methods of architecture components. In the video annotator, clicking down on any of the control buttons (*play*, *fast-forward*, etc.) highlights the button. In the UAN, we indicate that the forward button should be highlighted when depressed by writing "fwd_button!" in the *Interface Feedback* column (figure 8(a)). In the architecture, highlighting is implemented by the *button* component (figure 8(b)). When the *button* component receives the mouse button input (not shown), it changes the appearance of the button, providing feedback to the action of depressing the mouse.

In *Vista*, clicking the "fwd_button!" text in figure 8(a) highlights the button component of figure 8(b) which implements the appearance of the play button.

Linking UAN Interface State to ADTs

User actions can cause displays to change but they can also trigger state changes. In the UAN, these state changes are specified using pseudo-code in the "Interface State" column of the UAN table. States are implemented in Clock architectures as abstract data types (ADTs), which are attached to architecture components. The state that is referenced in the pseudo-code of the UAN tables can then be linked to the ADTs implementing the state.

For example, in figure 8(a), the UAN specifies that clicking on the *rev_button* should change the video command state to reverse ("videoCommand == reverse"). The video command state is implemented by the *VideoCommand* ADT located in the root of the architecture tree (figure 5). *Vista* implements links between interface state and ADTs. For example, clicking on the *videoCommand* text in figure 8(a) highlights the *VideoCommand* ADT.

This section has shown that four diverse artifacts from the user-centered design of user interfaces can be effectively linked. The links between design representations are complex, due to their differing points of view and differing levels of detail. The next section discusses how the ability to display these complex relationships aids in the coevolutionary design of user interfaces.

ANALYSIS

The last section showed how the *Vista* tool exposes the common elements of different design artifacts by linking these artifacts. In this section, we discuss how linking design artifacts supports the coevolutionary design of user interfaces. Our experience with *Vista* has been gained from four case studies: the development of a critical path planning application, the video annotator, a simple video player, and of *Vista* itself. Each of these case studies involved a group of people working on different aspects of the design and implementation, and each involved coevolution. Two of the applications are complex, two involve multimedia, and two are real-time groupware applications. We illustrate the benefits of links through examples from the video annotator application, and by discussing our experience in applying *Vista* to its own development.

Linking design artifacts helps designers collaborate in two ways: by helping to bridge gulfs of communication, and by adding value to the artifacts each designer uses.

Links aid communication among designers by showing how parts of one design artifact are reflected in others. This helps designers quickly locate parts of other design artifacts that are of interest to them, allowing them to access information in the artifact without having to read and understand it in its entirety.

We shall separately consider how links are helpful to HCI designers and to software engineers.

How Links Help HCI Designers

Linking design artifacts helps HCI designers by revealing the state of development, providing feedback on the implementation of the task specification, and by helping with the analysis of the complexity of the user interface design.

Reveal the State of Development

Linking the design artifacts developed by software engineers to those developed by HCI designers helps communicate the state of development to the HCI designers. HCI designers can see what features have been provided to implement their designs, and how far the implementation has progressed. Even if the HCI designer does not understand the details of an architecture, seeing its complexity gives the designer feedback on the feature's implementation cost. This feedback can aid discussions on whether user interface features should be redesigned to reduce the difficulty of their implementation.

For example, HCI designers can see that the task of navigating the video is implemented in components implementing a set of arrow buttons, and by the gestalt view.

Support Continual Refinement of Task Hierarchy

Task descriptions are not static entities, but instead evolve as understanding of users' tasks improves, and as the tasks themselves evolve as new technologies and processes are introduced in the workplace. Implementation helps in refining the understanding of how users perform their jobs, and how users will change their work patterns following the introduction of the system under development. Ideally, this information should be fed back to the HCI designers who perform the task analysis. Links between architecture and task-oriented specification can help facilitate this feedback.

We experienced this feedback in the development of the *Vista* tool itself. *Vista* was originally developed in order to demonstrate how links could be found between different design artifacts. The original task analysis underlying *Vista* paid little attention to how designers would actually use the tool to build systems. Once the first prototype had been built, it became evident that a tool of this form would be useful to developers. The task analysis was revised to better show how the tool would be used to support coevolution.

One concrete improvement to *Vista*'s design that resulted from this process was the identification that links need to be preserved as the structure of the architecture evolves. Architecture structure evolves significantly throughout coevolutionary design. The need to automatically maintain links throughout this evolution was not demonstrated in the original task analysis, but became clear when the second task analysis was developed. This in turn identified a significant flaw in the design of the *Vista* tool itself.

Indicate Complexity and Design Parsimony

Links between tasks and architecture components show what components are required to implement a task. The number of components gives some measure of the complexity of the task-action mapping. The links from task-oriented specification to architecture therefore augment the information present in the task-oriented specification, providing useful additional information to the HCI designer.

Similarly, linking architecture back to task hierarchy gives a measure of the parsimony of the user interface design, showing how many tasks are supported by a single component.

For example, the task hierarchy for the video annotator (figure 3) shows separate tasks for annotating a frame and for identifying that a frame is interesting. *Vista* shows that these tasks are in fact carried out by the same mechanism of annotating a frame, as implemented by the *Whiteboard* architecture component.

How Links Help Software Engineers

Linking design artifacts helps software engineers by helping to keep the perspective of the user visible throughout implementation. We found that links help keep design rationale visible and aid in implementation.

Keeps the Design Rationale Visible

Links between task hierarchy and architecture help to record design rationale. Links clearly show the implementer what tasks a user interface feature is meant to support. This helps the implementer assess the importance of keeping the feature as designed. In particular, user interface features may have been designed to support more than one task. In the heat of implementation, a software engineer may lose track of these multiple contexts of use, and optimize a feature for one kind of use. Links from architecture to task hierarchy avoid this problem by revealing to implementers all contexts of a feature's use.

For example, the gestalt view of the video annotator (figure 7) can be seen as an aid to navigation. The gestalt view requires a difficult custom implementation, since it cannot be based on standard widgets provided with user interface toolkits. An implementer might be tempted to substitute a different navigation mechanism that is easier to implement. The task hierarchy (figure 3) shows, however, that a major task of users is to find out what parts of a video are interesting. The gestalt view supports this task – wherever annotations form a cluster in the gestalt view, a segment of the clip is interesting. A different interaction technique might support the navigation function of the gestalt view, but not the analysis function of identifying interesting segments of the video. By linking the task-oriented specification to the architecture view, the two functions of the gestalt view are clearly shown.

Aiding Implementation

User interface implementation is complex, and can be greatly helped by the precise descriptions provided by UAN specifications. Our experience with programmers using UAN specifications has shown, however, that programmers complain about the length of UAN specifications, the distance of UAN specifications from their code, and the difficulty of locating relevant parts of UAN specifications [8]. Linking architecture and code to UAN specifications helps programmers find what parts of the task-oriented specification are relevant to the part of the system under development. The precise specifications are then readily available and the chances of forgetting important requirements are reduced.

Task-oriented specifications provide a reminder of the user's perspective. It is helpful to implementers to be able to quickly retrieve this perspective through *Vista*'s hyperlinks.

RELATED WORK

The work described in this paper draws on a wide body of related research in task analysis, task-oriented specification and software architecture. It is widely accepted that coevolutionary design is a natural and preferred design process [2,4,6,7,10,13,16].

The importance of task analysis in implementation is widely recognized [5]. Diaper [6] links task analysis with entity modeling by automatically generating entity life history matrices. Adept [14] is a tool for automatically generating user interfaces from task modeling artifacts which has been used in the domain of medical applications. In our earlier work, we showed how software architectures can be methodically derived from task-oriented specifications [8].

Vista depends on the inclusion of software architecture design in the development process. Software architecture provides a high-level abstraction for code that makes links more clear. In recent years, there has been substantial work in demonstrating the importance of software architecture in the development of interactive systems [3,9,12,17]. The architecture styles that have been proposed all have common elements: a hierarchical structure representing the compositional structure of the user interface, and a reliance on implicit invocation (through constraints or view triggers) for display updates. This convergence of basic features in user interface architecture styles suggests that *Vista*'s linking techniques could be applied to implementation languages other than Clock.

CONCLUSIONS

This paper has shown that design artifacts developed to produce user-interfaces can represent radically different views of a system under development. We have described two design artifacts that represent a user-centered perspective and an implementer's perspective of a system under development. We described the links between these two design artifacts in some detail. We then presented a tool called *Vista* that was developed to help designers explore design artifacts through these related links.

We presented a view of the development process of a system that suggests that design artifacts coevolve. We have shown how this is the case with the two design artifacts we focused on in this paper. We believe that coevolution of design artifacts can be supported by tools like *Vista* that visually display user-interface design artifacts and their links.

The work with Vista is ongoing. We are currently researching how links can be created and maintained in a more automatic manner than our current approach. A new Java-based version of Clock is being developed, which should allow Vista to be migrated to a more conventional programming language.

ACKNOWLEDGMENTS

The video annotator example [9] was programmed by Tore Urnes. This work was partially supported by the Natural Science and Engineering Research Council, the Information Technology Research Centre and Communications and Information Technology Ontario.

REFERENCES

1. Berry, R., Isensee, S. and Roberts, D. *OVID: Object View and Interaction Design,* Tutorial notes for CHI97.
2. Booch, G. *Object-Oriented Analysis and Design with Applications, 2nd ed.* The Benjamin/Cummings Publishing Company, Inc. 1994.
3. Coutaz, J., PAC: an object-oriented model for dialog design, *Proceedings of INTERACT'87,* 431-436, 1987.
4. Curtis, B. *…But You Have to Understand, This Isn't the Way We Develop Software at Our Company.* MCC Technical Report STEP-203-89. May 1989.
5. Diaper, D. Task observation for Human-Computer Interaction. In Diaper, D., editor, *Task Analysis for Human-Computer Interaction.* Ellis Horwood, 1989.
6. Diaper, D. Integrating HCI and Software Engineering Requirements Analysis. *SIGCHI Bulletin* 29, 1, 41-50.
7. Fisher, G., Redmiles, D., Williams, L., Puhr, G.I., Aoki, A. and Nakakoji, K. Beyond Object-Oriented Technology: Where Current Approaches Fall Short. *Human-Computer Interaction,* 10,1, 79-119, 1995.
8. Graham, T.C.N., Damker, H., Morton, C., Telford, E. and Urnes, T., *The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation.* York University Technical Report CS-96-04, 1996.
9. Graham, T.C.N. and Urnes, T, Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces, in *Proceedings of ICSE'97,* ACM Press, 172-183 ,1997.
10. Graham, T.C.N., Viewpoints Supporting the Development of Interactive Software, in *Proceedings of Viewpoints 96,* ACM Press, 263-267, October 1996.
11. Hartson, H.R., Siochi, A.C. and Hix, D. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems* 8, 3, ACM Press, 181-203, July 1990.
12. Hill, R.D., Brinck,T., Rohall,S.L. and Patterson,J.F. The *Rendezvous* language and architecture for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction* 1,2, 81-125, June 1995.
13. Jacobson, I. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
14. Johnson, P., Johnson, H. and Wilson, S. *Rapid Prototyping of User Interfaces Driven by Task Models in Scenario-Based Design: Envisioning Work and Technology in System Development.* John Wiley & Sons Inc., 1995.
15. Krasner, G. and Pope, T., A Cookbook for using the Model-View-Controller Interface Paradigm. *Journal of Object-Oriented Programming,* 1,3, 26-49, 1988.
16. Parnas, D.L. and Clements, P.C., A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering* SE-12,2, 251-257, Feb. 1986.
17. Taylor, R.N., Nies,K.A., Bolcer,G.A., MacFarlane,C.A., Anderson, K.M. and Johnson, G.F. 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.