Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces

T.C. Nicholas Graham **Tore Urnes**
Department of Computer Science
York University
4700 Keele St., North York
Canada, M3J 1P3
+1 (416) 736-5053
{graham, urnes}@cs.yorku.ca

ABSTRACT
As computers are increasingly used for communicating information, multimedia has become an important component of interactive applications. Effective communication with multimedia requires tight media integration, highly interactive control over multimedia, and the use of multimedia to support group interactions. This paper shows how the MVC paradigm for user interface development can be extended to support temporal media. The resulting framework allows easy specification of media-integrated interactive applications, including multimedia groupware. All examples in this paper have been implemented in Clock, a novel programming environment, and run on IBM, SGI, and Sun workstations.

Keywords
Multimedia programming, MVC, software architecture, groupware

INTRODUCTION

Computers are increasingly being used as tools for communication, supporting the dissemination of information and aiding people in collaboration over a distance. In addition to the traditional static media of text and graphics, the temporal media of sound, video, music and animation have become increasingly important in improving the quality of communication supported by user interfaces.

Programming multimedia applications, particularly those involving multiple users, has traditionally been a complex task. Most of the problems of multimedia programming relate to its temporal nature — sound and video clips have duration, requiring the programmer to orchestrate the concurrent playout of multimedia streams and to synchronize them with the views and actions of multiple users. It is difficult to integrate temporal media with static media, for example associating a drawing with a particular frame of a video, or making a particular object in a video sensitive to input. These difficulties have led to limited use of multimedia in a multiuser context, and limited integration of multiple media within user interfaces. Since user interface development already accounts for 50% of the cost of software development [17], it is important to explore how software engineering tools can aid the integration of multimedia into graphical user interfaces.

This paper demonstrates how the use of a high-level software architecture based on the model-view-controller (MVC) architecture [14] can aid in the development of multiuser, multimedia applications. We present an extension to MVC that removes the temporal problems from multimedia programming, allowing easy synchronization of multimedia over multiple users, and easy integration of temporal media with static media. This architecture has been implemented in the Clock groupware development toolkit which runs on UNIX workstations.

In order to motivate why programming multimedia applications is difficult, an example multimedia application is presented. The key properties of the MVC architecture are then introduced, and illustrated using the Clock groupware development toolkit. Our earlier work [10, 11] showed how Clock's composite MVC architectural style is appropriate for the development of highly interactive groupware systems. This paper shows how this architectural style can be extended to support multimedia. The following sections show how support for temporal media is a natural extension of the MVC architecture, and discusses how this high-level architecture can be mapped to concrete implementations.

Example Multimedia Application

To illustrate the problems of programming multiuser, multimedia applications, we introduce an example of a collaborative video annotator. This application allows a group of users to simultaneously annotate a video as it is playing. As shown in figure 1, multiple users can add annotations and view the annotations of other users as they are made in real time. A video annotator could be used to aid in distance learning or for collaborative

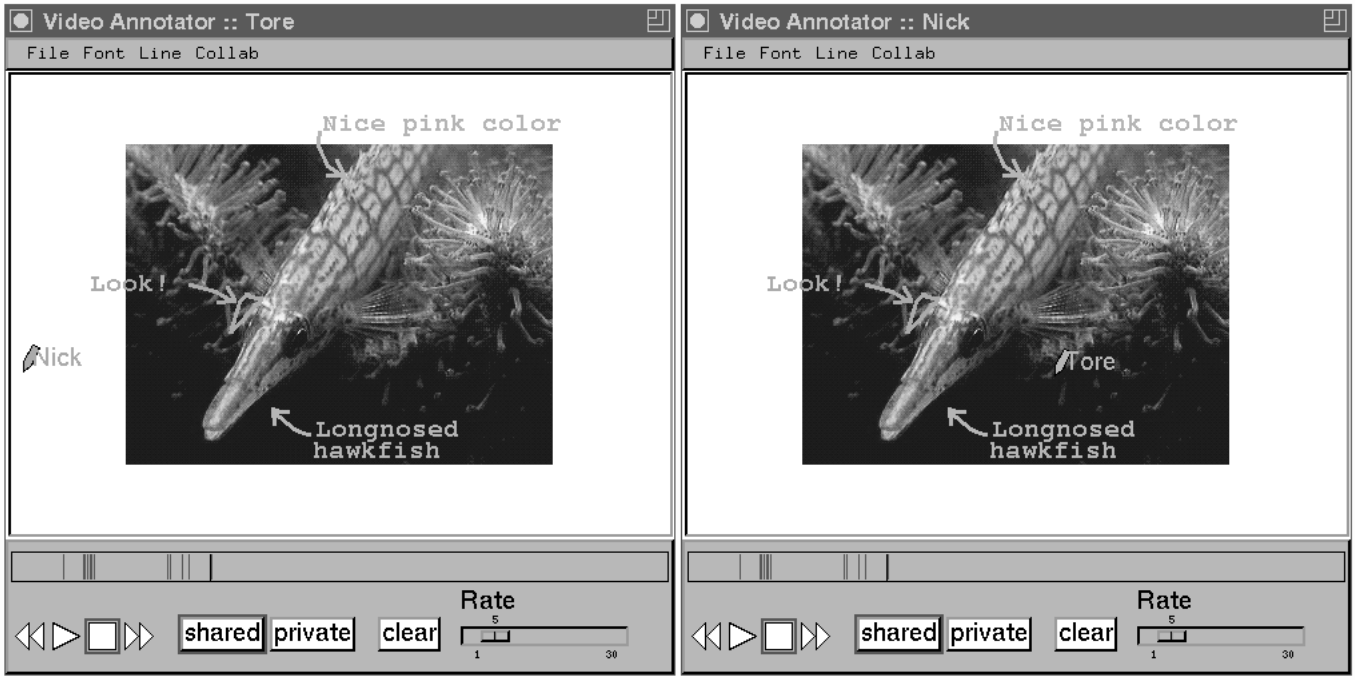

Figure 1: A multimedia application: collaborative video annotation

analysis of the contents of a video. The annotator of figure 1 was implemented in Clock, and runs on SGI and Sun workstations.

We first briefly describe the user interface of the collaborative video annotator before explaining some of the difficulties in implementing this class of application.

The main component of the user interface shown in figure 1 is a shared whiteboard. The whiteboard allows users to simultaneously make sketches and type in text. The video being studied is played back in the background of the whiteboard. Annotations may be drawn on top of the video literally as it is playing. Subsequent playouts of the video will include the annotations.

A shared video control panel appears below the whiteboard in figure 1. It offers standard VCR controls in addition to a scale for setting the playout rate. A customized scale widget is provided for directly jumping to positions in the video playback. In addition to reflecting the playout position, the scale also provides a simple time-line chart detailing the positions of annotations made so far. A *Clear* button erases all annotations from the current frame.

The top portion of the user interface contains a private menu bar that offers a range of commands allowing users to customize the operation of the annotator, including setting font and line sizes and turning telepointers on or off.

Finally, interaction mode radio buttons allows selection between *shared* and *private* operation. In shared interaction mode, when a user commences an annotation, the playout stops for all users, and all users can contribute to the annotation of the current video frame. When a user starts to annotate a video frame in the private interaction mode, however, the playout seen by the other users continues uninterrupted. Even when a private annotation is made, other group members can be aware of the annotation through its appearance in the time-line chart

This example illustrates some of the issues that need to be tackled in interactive multimedia applications. First, video images have a prominent role in the user interface. It must be possible to integrate them seamlessly into the interface and to allow overlapping of annotations, menus, and telepointers. Second, substantial interactive control over the video playout is required. It must be possible to pause, play backwards, fast-forward, and jump to an arbitrary position in the video should any of the users desire to do so. Similarly, users need to have full control over the playback speed. Third, it is often beneficial to have multiple users collaborate on analyzing a video. Therefore, it is required that annotated video views be kept consistent across the interfaces of all users without sacrificing responsiveness. Below, we will show how these issues are tackled by our MVC-based

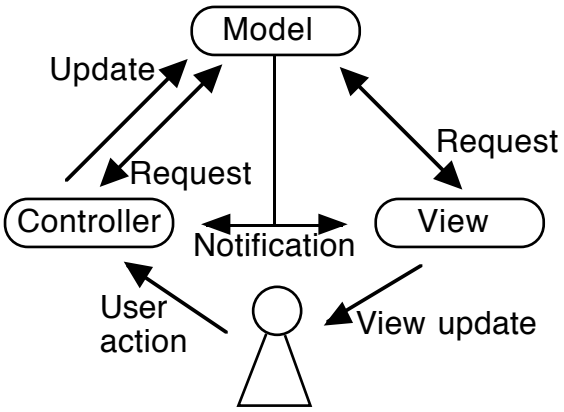

Figure 2: The Model-View-Controller Architecture for organizing interactive applications.

approach to supporting multimedia programming.

The MVC Architecture

Variants of the model-view-controller architecture have been widely adopted in the user interface community as a high-level way of organizing user interfaces [1, 16, 12]. Advantages of MVC and its derivatives include separation of concerns, simplified maintenance, and ease of evolution. Below, we shall see that by extending MVC to handling temporal media, these advantages can extend to providing easy media integration and easy synchronization of the multimedia interfaces of multiple users.

As seen in figure 2, the MVC architecture partitions interactive applications into three separate parts: the *model* implements the application's data state and semantics; the *view* is responsible for the graphical output of the application, and the *controller* takes care of interpreting inputs.

An important aspect of the MVC paradigm is the set of rules governing how model, view, and controller communicate [14]. The controller transforms *user actions* into *updates* that are sent to the model. The updates cause the model to change its data state. Changes to the model, in turn, result in *notifications* being sent to the view and controller. When notified about a state change in the model, the view starts computing the display. During computation, the view will issue *requests* to the model to query the new data state. The controller may also need to request data from the model when transforming user actions.

A key consequence of the MVC communication rules is the absence of a notion of time in the view. The view encoding never makes decisions about when displays need to be updated. Instead, display updates are

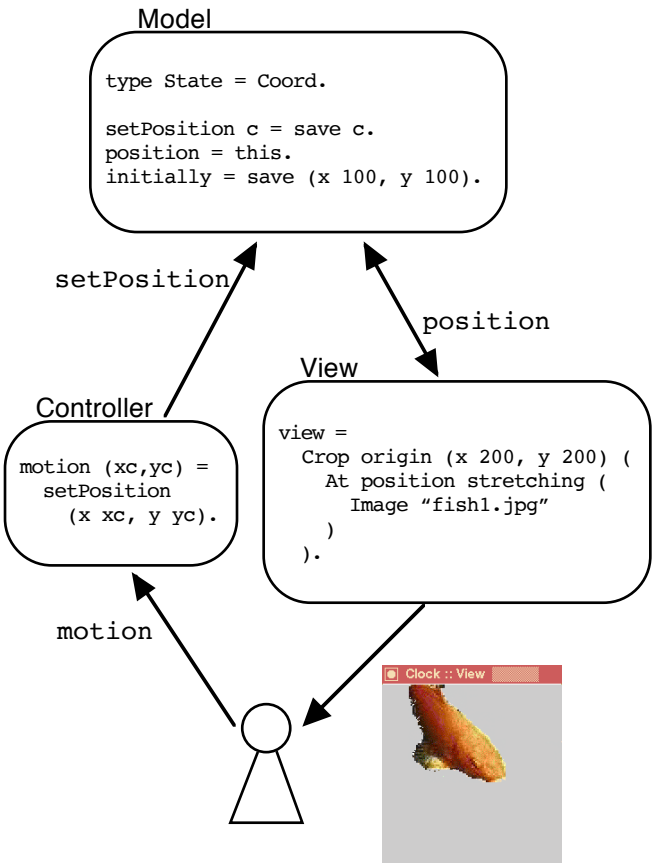

Figure 3: Clock code specifying that the image of a fish should follow the mouse pointer inside a canvas. The code is organized according to MVC.

simply performed whenever changes in the model's data state make them necessary. As will be seen below, the key property satisfied by our extension to MVC to temporal media is that the view must never have to make any temporal decisions.

The Clock Programming Environment

Throughout this paper we will be presenting example multimedia applications developed under the Clock programming environment. As an aid to better understand the examples, the following briefly introduces Clock. Clock is one possible realization of a programming environment based on MVC.

The Clock Language

Clock allows interactive applications to be written using a functional language with syntax similar to that of Haskel [13]. Figure 3 shows the complete Clock code of a very simple interactive application. It presents the user with a canvas containing the image of a fish. If the user moves the mouse pointer inside the canvas, the fish image will follow it. The Clock code is partitioned

between the model, view, and controller.

In Clock, *controllers* are specified as a set of declarative rules that transform user actions into updates to the model. In figure 3, a single rule states that mouse `motion` is to result in `setPosition` updates. The *model* in our simple example consists of an ADT that encapsulates a single value: the current position of the mouse pointer. The ADT implements two methods: an update and a request. The update (`setPosition`) replaces the old position by the given new one. The request (`position`) returns the current position.

A *view* is specified by a `view` function. The result of evaluating a `view` function is always a picture. In the example, the `view` function returns a picture of some object (`Image ...`) at some position (`At ...`) confined to be inside a 200-by-200 rectangular area (`Crop ...`). The position of the object is obtained from the model by issuing the `position` request. Note that when the view requests the current position of the object, it also implicitly creates a constraint between the data state of the model and the display of the view. This constraint ensures that whenever the position in the model changes, the view will be automatically triggered to bring the display up to date.

This constraint mechanism removes the notion of time from MVC views. The programmer does not need to aware of when the view will be computed, and simply programs with the guarantee that the view will be invoked whenever necessary (i.e., whenever `position` changes). As we shall see in the next section, this removal of time from views is the cornerstone of providing high-level support for temporal media within MVC.

The Clock Architecture

According to MVC as presented above, applications must be divided into exactly three monolithic parts. Interactive applications typically have a structure that is too complex to represent using such a simple architecture. It is therefore standard to extend the architecture to permit hierarchical composition of MVC components. Composition offers the benefits of flexibility and better support for reuse [16].

In Clock, applications are structured as a tree of model-view-controller clusters. The tree is specified using a visual notation. We call this tree the *architecture* of the interactive application. The MVC clusters from which the architecture is composed are called *components*.

An example architecture consisting of two components is shown in figure 4. This example is a somewhat extended version of the example in figure 3. It presents the user with a canvas containing three fish images. The user can interact with each fish image independently by clicking on it with the mouse pointer and moving it around inside the canvas.

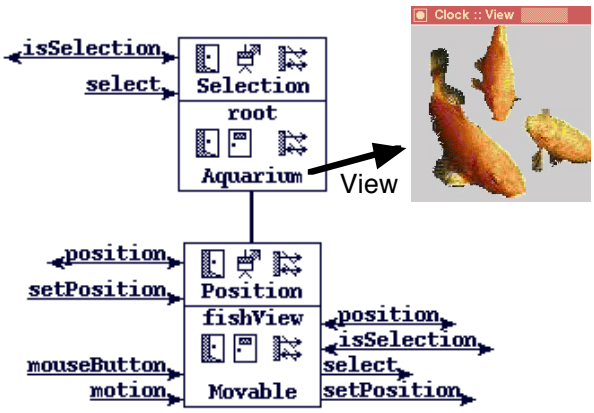

Figure 4: The architecture of a simple interactive application where the user can interact (click and move) with fish images. It illustrates simple composition of MVC clusters. The `Aquarium` view-controller has a model consisting of a `Selection` ADT capable of handling an update (`select`) and a request (`isSelection`). The `Movable` view-controller takes `motion` and `mouseButton` inputs and issues several updates and requests.

In the visual notation, a component is depicted as a set of boxes. The bottom box represents a view-controller pair. It has two labels: the component's class name and subview name. In figure 4, the root component has class name `Aquarium` and subview name `root`. Any boxes stacked on top of a view-controller pair represent ADTs making up the component's model. In figure 4, the model of the root component consists of one ADT: `Selection`. The requests and updates issued by a component are depicted on the right hand side of the component in question. Requests, updates, and user actions that a component is capable of handling are shown on the left.

In Clock, composition is overloaded: we structure the architecture tree both to reflect the composition of the user interface and to reflect the inheritance (by delegation) relations between the component classes. The user interface shown in figure 4 is composed of a canvas (`Aquarium`) containing three objects (`Movable`).

Each component in the architecture tree represents a class that may be instantiated at run time. A parent component determines how many instances should be created of each of its child components. Therefore, the `Aquarium` component will instantiate the `Movable` component three times. It does this by invoking the child component's subview name (`fishView`) in its view function. Here is the view function of the `Aquarium` component:

```
view =
  Crop origin (x 200, y 200) (
    Views [
      fishView "fish1",
      fishView "fish2",
      fishView "fish3"
    ]
  ).
```

The invocation `fishView "fish1"` will create an instance of `Movable` with the id "fish1" (if one did not already exist) and return the picture resulting from the child's view function. Pictures from children components can therefore be used to create the parent component's view (the `Views` command creates a new picture by drawing a list of pictures on top of each other).

The Clock programming environment provides a graphical editor, *ClockWorks* [9], allowing architectures to be built, browsed, and edited using the graphical notation.

Clock's compositional architecture structure and use of constraints to implement MVC communication is similar to the approaches of the PAC architecture [1] and the Rendezvous ALV architecture [12]. There are also many similarities between Clock's architectural style and the Chiron-2 architecture [20]. Our extension of MVC to handle temporal media should therefore be directly applicable to numerous systems other than Clock.

ADDING TEMPORAL MEDIA TO MVC

The previous section has shown how MVC provides a simple, high-level architecture for organizing interactive applications. In particular, MVC frees the programmer from having to consider when and how views are to be updated, leaving this decision to the run-time system. This absence or temporal issues in specifying views also aids in the programming of multimedia applications involving temporal media. This section shows how adopting a *pools of frames* model of multimedia allows us to exploit the MVC architecture for handling temporal media, allowing easy integration of temporal and static media, and easy synchronization of the views of multiple users. The following section describes how this model can be efficiently mapped to an implementation architecture.

The most common model for processing multimedia data is the *stream model* [3, 4], based on a time-ordered flow of information from a source to a target. The main benefit of the stream model is modularity. Not only are producer and consumer of multimedia data decoupled, but stream processing elements may be arbitrarily interconnected. The main drawback of the stream model is that it is highly asynchronous. Typically, a stream is implemented as a queue of data buffers, causing substantial, variable delays from source to target. This makes it difficult to provide random access to the media data and to tightly integrate multimedia into interactive applications. Also, the notion of a time-ordered flow makes it difficult to incorporate streams into MVC without violating the temporal independence of views.

Media as Pools of Frames

Numerous media types can be incorporated into interactive applications, including text, graphics, images, audio, video, music, and animation [6, chapter 2]. The latter four media types are often known as temporal media. By treating temporal media as *pools of frames*, we can model them within the MVC framework.

A video clip can be viewed as a pool of video frames where each video frame is simply an image. Similarly, an audio clip is a pool of audio frames where each frame is a small audio segment which in turn consists of a sequence of samples. A frame is something that can be mapped to an output device and perceived by the user. The frame abstraction is a simple concept and can be applied to all media types.

A pool of frames has no embedded notion of implicit temporal constraints (the notion of continuous flow). Instead, each frame is assigned a unique frame number, imposing total ordering over all the frames in the pool. Frame numbers normally reflect the order in which frames where recorded. Frame numbers make it easy to randomly access frames in the frame pool through primitives that, given a frame number and a frame pool, extract the correct frame and map it to an output device. Therefore, playout of temporal media is reduced to producing a sequence of frame numbers in a timely fashion. Temporal issues are no longer inherent in the media data themselves.

Figure 5 shows an example of how simple playout of a pre-recorded video clip is realized using MVC and our frame-pool temporal media extension. The complete Clock code is given.

The current frame number is stored in the model. In the example, the model defines an update (`increment`) which changes the data state by adding one to the current frame number. It also defines a request (`count`) which returns the current frame number.

The controller is able to handle a new input event, the `tick` input, in addition to user inputs. The `tick` inputs are generated be the Clock run-time system at some regular frequency (the frequency is under programmer control, with a default of 30 ticks per second). The controller in figure 5 specifies that every `tick` input should be transformed into an `increment` update.

In the view, a `VideoFrame` primitive is used to extract a given frame from a frame pool and return the corresponding image. The frame pool is stored in the file

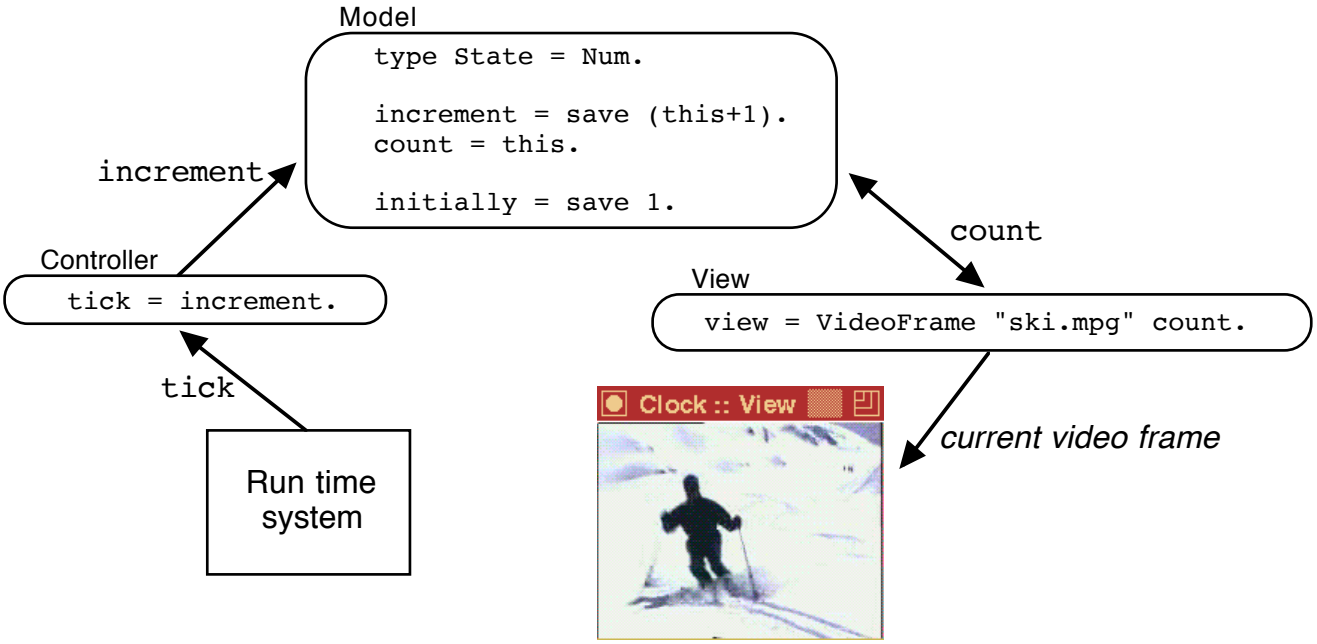

Figure 5: Complete Clock code for a simple program that plays out a pre-recorded video clip. The code is organized according to the MVC model with extensions for temporal media.

`"ski.mpg"` and is encoded using the MPEG compression standard. Every time the view is notified about a state change in the model, it uses the `count` request to query what the current frame number is before updating the display.

In other words, `tick` inputs and the `increment` update cause the model to go through a sequence of frame numbers in a timely fashion. The view specification automatically translates this frame number sequence into a playout of the media clip in question.

This example shows how splitting temporal media clips into frame pools allows them to be used within an MVC architecture. The projection of temporal clips onto frames allows them to be treated as static media (e.g., images or sound bites). Next, we show how the frame pool approach leads to easy media integration.

Media Integration

The primary focus of media integration is how digital video can be integrated with interactive user interfaces. Existing applications that feature video typically treat it as something outside of the actual user interface. We believe that one reason for this is that most existing support for multimedia programming, i.e. stream-based approaches, makes it very hard to mix video with other media. Here, we seek to demonstrate how our MVC-based approach to multimedia programming solves this problem.

The collaborative video annotator in figure 1 is an example of an application that tightly integrates video with other media. Users can draw and type in text directly on video frames and on the surrounding whiteboard. These drawings effectively become part of the video, and can be retrieved when the video is rewound to the annotated frame. We now show how the pool of frames approach under MVC supports this media integration.

Figure 6 depicts the Clock architecture of the collaborative video annotator application. As usual, the Clock architecture reflects the composition of the user interface. Each user in a collaborative video annotator session is provided with a window containing a user view. The user view is made up of a menu canvas and some telepointers. Inside the menu canvas there is a workspace consisting of a control panel (elided) and a whiteboard. Finally, the whiteboard incorporates video and annotations.

In figure 6, all the key ADTs are kept in the model of the root component. They include a frame counter (`Counter`) identifying the current frame of the video, the annotation database (`AnnotationDB`) maintaining the annotations for each frame, the coordinates of all the telepointers (`TeleCoord`), and the session database (`SessionDB`) with information about the users currently in the session. The controller of the root component translates clock ticks into updates to the frame counter in a similar fashion to figure 5.

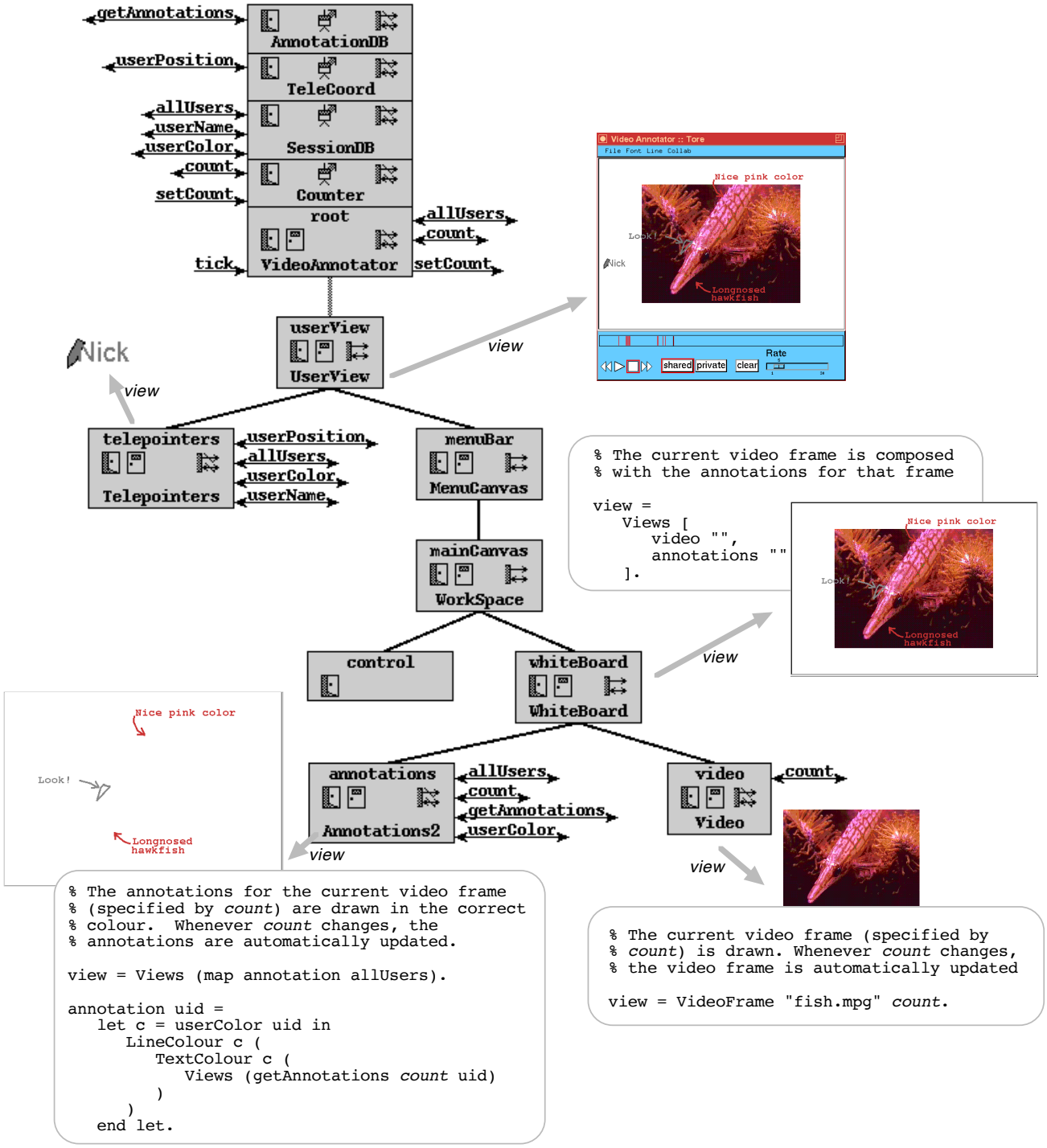

Figure 6: The architecture of the collaborative video annotator of figure 1. The extended MVC model allows temporal and non-temporal media to be easily mixed. Note that parts of the architecture has been elided and that only essential updates and requests are shown.

The key to simplifying the integration of video with other media is the decoupling of time from temporal media as facilitated by the extended MVC paradigm. It allows us to reduce media integration to simple view composition. Clock offers a 2.5 dimensional declarative graphics system that makes view composition particularly simple. Figure 6 shows how the tight integration of video and annotations inside the whiteboard (the `WhiteBoard` component) is accomplished through a few simple view specifications.

The view functions of the `Video` and `Annotations` components are, respectively, responsible for displaying the current video frame (as an image) and the annotations associated with that frame (as a set of lines and texts). Both of these views depend on the current frame number (`count`), so that whenever `count` is changed, the views of the video frame and annotations are automatically updated. The `WhiteBoard` component composes these two views (via the `Views` primitive), producing a mixed-media annotated video image.

The removal of temporal issues from the views implementing the video annotator is therefore the key to easy media integration. The views of the `Video` and `Annotations` components are both simply pictures, which can be combined in the `WhiteBoard` component. The underlying MVC architecture guarantees that as the frame counter changes, both the video frame and the annotations will be updated, transparently to the whiteboard.

Interacting with Temporal Media

In addition to producing mixed-media output, it is also important to be able to interact with user interfaces involving multimedia. For example, in the video annotator, users may stop and restart playout, reverse, and jump directly to any point in the video stream. Additionally, users may dynamically change the frame rate during playout by manipulating the frame rate scale.

More media-specific ways of interacting are also possible. The images constituting the playout of a video might, for example, be dynamically moved and resized by the user. For audio playouts, the gain (or volume) is an obvious candidate for user manipulation.

The MVC approach allows all of these interaction styles to be programmed in a natural way. For example, in the video annotator, the display depends on the current frame counter. Playing forward (as we saw in figure 5) consists of regularly incrementing the frame counter in response to tick inputs. In a similar way, playing in reverse simply means that tick inputs should *decrement* the frame counter; stopping the video means that tick inputs should be discarded (i.e., not modify the frame counter at all).

The rate at which tick inputs are generated by the run-time system is under programmer control (via a `TickingEvery` primitive); a `Gain` primitive allows control over volume.

Shared Playout

Any treatment of multimedia as a tool for communication must consider how multimedia can be used to help people communicate with each other. For example, the video annotator permits groups of people to analyze and discuss videos in real time, even if they are located at different sites.

The MVC architecture is useful for structuring multiuser applications, including those involving temporal media. According to the MVC paradigm, the model has no knowledge of the view and controller other than the fact that they are depending on its data state and need to be notified about changes. The MVC architecture may therefore easily be extended to support multiple users by associating multiple view-controller pairs with a single (shared) model [12]. Composition further provides facilities to easily maintain both shared and private data in applications like the collaborative video annotator. In Clock, the root of the architecture tree serves as a shared model while each user receives a private instance of subtrees below the root.

By simply representing frame numbers in a shared model, it is possible to synchronize clip playout across the displays of a group of users. The shared video playout of the collaborative video annotator is implemented in this way: as seen in figure 6, the frame counter is represented in the root `VideoAnnotator` component. It is therefore shared by all users. The frame counter can be updated by users manipulating it directly through various widgets, or indirectly through the *tick* inputs. Any update to the frame counter will automatically trigger the views of all users.

This section has shown how an extended MVC architecture supports the development of multimedia user interfaces, providing easy integration of static and temporal media, high control over the interactive behavior of the application, and easy synchronization of the views of multiple users. An obvious question is whether such a high-level architecture can be implemented efficiently enough to satisfy the performance demands of interactive software. The next section details our implementation of the Clock groupware development tool, and shows how good performance can be obtained.

IMPLEMENTATION

The previous sections presented the high-level architecture and programming model of the Clock groupware development tool. This section describes the low-level implementation architecture of the Clock run-time system, focusing on how temporal media are supported.

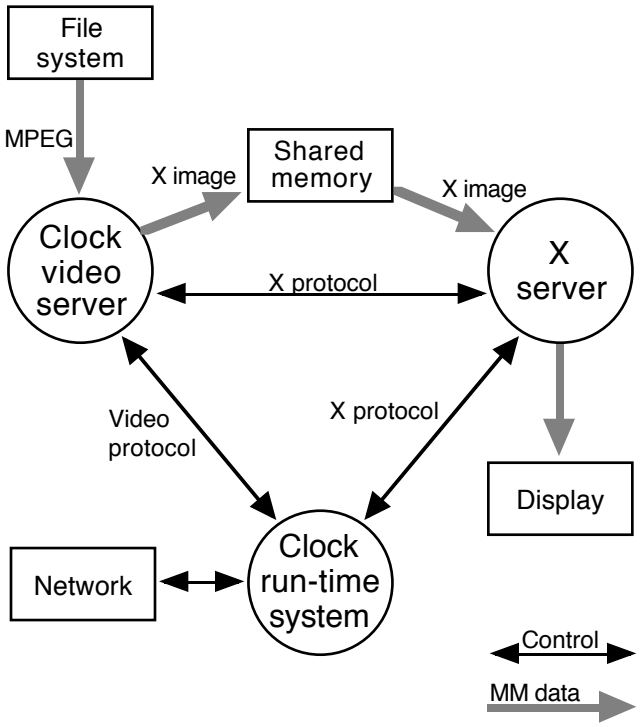

Figure 7: Implementation architecture for the video processing parts of the Clock system. Thick arrows depict the flow of video data.

Though programmers do not need to be aware of the workings of the implementation architecture, this presentation shows how Clock programs are implemented behind the scenes, and provides a basis for discussing the performance of this approach.

The Clock implementation architecture supports both video and audio. Note, however, that this section will only describe the parts devoted to handling video, leaving out everything related to audio. The organization of the Clock implementation architecture is depicted in figure 7.

The implementation architecture consists of the Clock run-time system, Clock video servers, and X servers [18]. The *Clock run-time system* realizes the high-level architecture. It implements the MVC communication rules, maintains clocks for tick inputs, orchestrates view computations, and takes care of input processing. A *Clock video server* realizes the frame pool model for video clips while *X servers* provide a device-independent interface to graphics hardware on a range of workstations.

During view computation, the Clock run-time system issues X and video commands to the respective servers. The video server reads MPEG-encoded video data from files stored in the local file system and transforms them into X images that are passed on to the X server. The following describes the Clock video server in more detail.

Video Server

The Clock video server is partly based on the software MPEG decoder of the X-Movie system [15]. It was extended to support parallel decoding of multiple MPEG-encoded video clips, random frame access, limited colormap resources, and multiple X servers. As indicated in figure 7, the video server uses shared memory for transmitting decoded images to the X server if the two are running on the same workstation.

The main principle behind the design of the video server and its protocol is that a request for posting a particular video frame should always be handled immediately, whether the frame in question is decoded and available or not. The server always tries to predict what the next frame posting request will be, based on past requests. It also keeps at least two of the most recently posted frames in a cache for each clip. In those cases when the video server failed to guess which frame to post next and the requested frame is not in the cache, the last posted frame is reposted. Once the requested frame has been decoded, the run-time system is notified that the desired frame is ready for posting. The run-time system may then decide to update the view.

Typically, when updating views that contain a mixture of video and other media, the run-time system will alternate between making X and video calls. This could be problematic because X calls are normally buffered at the client side (i.e., by the run-time system) and the video server will typically respond to video calls by issuing X calls on its own. Since the ordering of X calls is significant, undesirable effects are likely to occur. This problem is solved by explicitly flushing all X-command buffers and restricting the video command protocol to be synchronous.

Performance

Clock's multimedia architecture differs from that of most multimedia implementations in that Clock's audio and video servers provide access to frames on request, rather than actively providing frames in sequence. This architecture is necessary to provide the flexibility of stopping, playing in reverse, or jumping directly to a specific frame.

When playing frames in sequence, there is next to no performance penalty in Clock's approach. The video server eagerly decodes frames so that they are available when requested. When accessing frames out of sequence, there can be a significant delay (e.g. 500–1,000ms) before the frame is displayed. This delay can result when the server has to decode several frames in order to obtain the desired frame. This is largely a problem of using a software decoder, and will likely

be solved as hardware MPEG encoder/decoders become more widely available.

Distribution
The network in Figure 7 indicates that Clock applications can be mapped onto a distributed implementation architecture. This allows us to provide faster responses in collaborative applications where multiple, remote users are interacting simultaneously [11]. Note that the current implementation does not provide support for transmitting multimedia across networks. Therefore, collaborative multimedia applications such as the video annotator presented above are best implemented using a distributed run-time system communicating with video servers running locally at the users' workstations. We are, however, currently working on incorporating support for ATM networking and media sources like cameras and microphones into Clock. Our hope is to support real-time multimedia distribution in the near future.

RELATED WORK

The focus of the work presented in this paper is on media integration and how to provide high-level development support for media-integrated user interfaces. The need for tighter integration of temporal media with user interfaces has been addressed by others before us. Gibbs et al. [5] provide a class library for realizing video widgets and actors. Their approach relies heavily on analogue hardware and the actual media integration is handled outside of the programming model. Schnorf [19] describes a project integrating support for video into the ET++ class library and application framework. Video is drawn into the frame buffer by external autonomous sources. A clever scheme involving multiple types of clip masks and normal drawing operations having side effects that update video clip masks allows smooth integration of video into the lower graphics layers of ET++.

A key characteristic of our approach to supporting the programming of multimedia is the decoupling of temporal issues from the actual media data. The temporal decoupling of time from media in multimedia programming models has previously been used in the Tactus toolkit [2]. The focus of the Tactus toolkit, however, is not media integration but rather to provide abstractions on top of a time advance implementation model that ensures timely playout of temporal media in a high-latency, distributed setting.

As already pointed out, *streams* constitute the most common model for multimedia programming [3, 4]. It is inherent in the stream model that temporal media are continuously flowing through the application, typically through pipelines of processing elements implemented as active objects. The flow is initiated by sending "start" methods to all the active objects; "stop" methods terminate the flow. In other words, streams encapsulate many temporal issues.

Unfortunately, it is difficult to support tight media integration with streams. Streams typically make it cumbersome to randomly access frames and to respond to interaction. Also, as demonstrated in [19], it can be quite challenging to integrate support for video streams into the lower graphics layers of toolkits. That is, streams are best suited for stand-alone basic playout that is largely pre-computed, e.g. where non-deterministic interactions are limited to users perhaps pressing a pause button.

Our high-level treatment of time in multimedia programming builds on the formal basis of the Clock language. Further information on the formal treatment of time in Clock can be found in [8], and its application to user interface development in [7].

CONCLUSION

This paper has demonstrated how support for temporal media is integrated into the Clock system for the development of synchronous groupware. The key to this integration is an extension of the MVC paradigm for user interface development. The extension retains the central MVC concept that views should be free of temporal information. This allows video and sound clips to be treated uniformly with text and graphics, permitting truly integrated multimedia groupware applications. Since MVC forms the basis of many modern user interface development tools, we believe that this approach should be widely applicable.

ACKNOWLEDGMENTS

Clock and *ClockWorks* were developed by the authors, Catherine Morton, Roy Nejabi and Gekun Song. This work was carried out at the York Laboratory for Computer Systems Research, and was partially supported by the Natural Sciences and Engineering Research Council, the Information Technology Research Centre, and the Royal Norwegian Research Council.

REFERENCES

[1] J. Coutaz. The Construction of User Interfaces and the Object Paradigm. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP '87, Paris, France), Published as Lecture Notes in Computer Science, No. 276, Springer-Verlag, pages 121–130, 1987.

[2] R.B. Dannenberg, T. Neuendorffer, J.M. Newcomer, D. Rubine, and D.B. Anderson. Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia. *Multimedia Systems*, 1(2):77–86, 1993.

[3] R.B. Dannenberg and D. Rubine. A Comparison of Streams and Time Advance as Paradigms for Multimedia Systems. Technical Report CMU-CS-94-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, March 1994.

[4] S. Gibbs. Composite Multimedia and Active Objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '91), ACM SIGPLAN Notices, Vol. 26, No. 11, pages 97–112. ACM Press, 1991.

[5] S. Gibbs, C. Breiteneder, V. de Mey, and M. Papathomas. Video Widgets and Video Actors. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '93, Atlanta, GA, USA, Nov. 3–5), pages 179–185. ACM Press, 1993.

[6] S.J. Gibbs and D.C. Tsichritzis. *Multimedia Programming: Objects, Environments and Frameworks*. ACM Press /Addison-Wesley , ISBN 0-201-42282-4, 1994.

[7] T.C.N. Graham. Constructing User Interfaces with Functions and Temporal Constraints. In B.A. Myers, editor, *Languages for Developing User Interfaces*, chapter 16, pages 279–302. Jones & Bartlett Publishers, 1992.

[8] T.C.N. Graham. *Declarative Development of Interactive Systems*, volume 243 of *Berichte der GMD*. R. Oldenbourg Verlag, July 1995.

[9] T.C.N. Graham, C.A. Morton, and T. Urnes. ClockWorks: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages & Computing*, 7(2):175–196, June 1996.

[10] T.C.N. Graham and T. Urnes. Linguistic Support for the Evolutionary Design of Software Architectures. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE 18, Berlin, Germany, Mar. 25–29), pages 418–427. IEEE Computer Society Press, 1996.

[11] T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '96, Seattle, WA, USA, Nov. 6–8), pages 1–10. ACM Press, 1996.

[12] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner. The Rendezvous Language and Architecture for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

[13] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell (v1.1). Technical Report YALEU/DCS/RR777, Yale University, August 1991.

[14] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[15] B. Lamparter and W. Effelsberg. X-MOVIE: Transmission and Presentation of Digital Movies under X. In R.G. Herrtwich, editor, *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video* (Published as Lecture Notes in Computer Science, No. 614, Springer-Verlag, 1992), pages 328–339, November 1991.

[16] M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing User Interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[17] B.A. Myers and M.B. Rosson. Survey on User Interface Programming. In *Human Factors in Computing Systems: CHI '92 Conference Proceedings* (Monterey, CA, USA, May 3–7), pages 195–202. ACM Press, 1992.

[18] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[19] P. Schnorf. Integrating Video into an Application Framework. In *Proceedings of ACM Multimedia '93* (Anaheim, CA, USA, Aug. 1–6), pages 411–417 & 478. ACM Press /Addison-Wesley, 1993.

[20] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., and J.E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering* (ICSE 17, Seattle, WA, USA, Apr. 24–28), pages 295–304. IEEE Computer Society Press, 1995.