The Clock Language

Preliminary Reference Manual

T.C. Nicholas Graham
York University
graham@cs.yorku.ca

Abstract: This document serves as preliminary documentation for the Clock language. Clock is a purely declarative language supporting the development of highly interactive graphical user interfaces. For now, this document concentrates on the syntax of the language and the predefined functions the language provides. The document is incomplete, and will continue to evolve.

Acknowledgements

Clock was designed and implemented by T.C. Nicholas Graham and Tore Urnes, with substantial contributions by Catherine A. Morton and Roy Nejabi. Other contributers to the Clock project have been Herbert Damker, Gekun Song and Eric Telford.

We gratefully acknowledge the financial support of the ITRC, NSERC, and the Royal Norwegian Research Council.

Overview

Clock is a declarative language intended for the programming of interactive systems. Features of Clock include:

Support for Multimedia

Clock provides high-level support for the traditional media of text and graphics, as well as the continuous media of sound and video.

Support for multiple users

In Clock, programming a multiuser application is almost as easy as programming a single user application. Support is provided for automatic distribution of programs over a network of workstations. Built-in concurrency control helps arbitrate between the concurrent actions of multiple users. High-level support is provided for shared and private views in different users' user interfaces, allowing easy programming of relaxed WYSIWIS (what you see is what I see) user interfaces.

Visual architecture language

The architecture (or high-level design) of Clock programs is carried out in the visual ClockWorks programming environment. ClockWorks makes it easy to visualize and modify the structure of complex applications. The architecture consists of components written in a high-level scripting language.

Simple, constraint-based view updates

Like most modern user interface toolkits, Clock provides constraints to permit automatic updating of displays in response to user actions. Constraints simplify the programming of interactive software by reducing the direct dependencies between program components. Clock's constraints are encoded as functions.

Flexible support for incremental and iterative design of applications

Clock provides high-level support for incrementally developing applications, and for quickly modifying applications. This support comes from the use of constraints to connect application components, and from the high-level support for modifying applications provided in the visual ClockWorks programming environment.

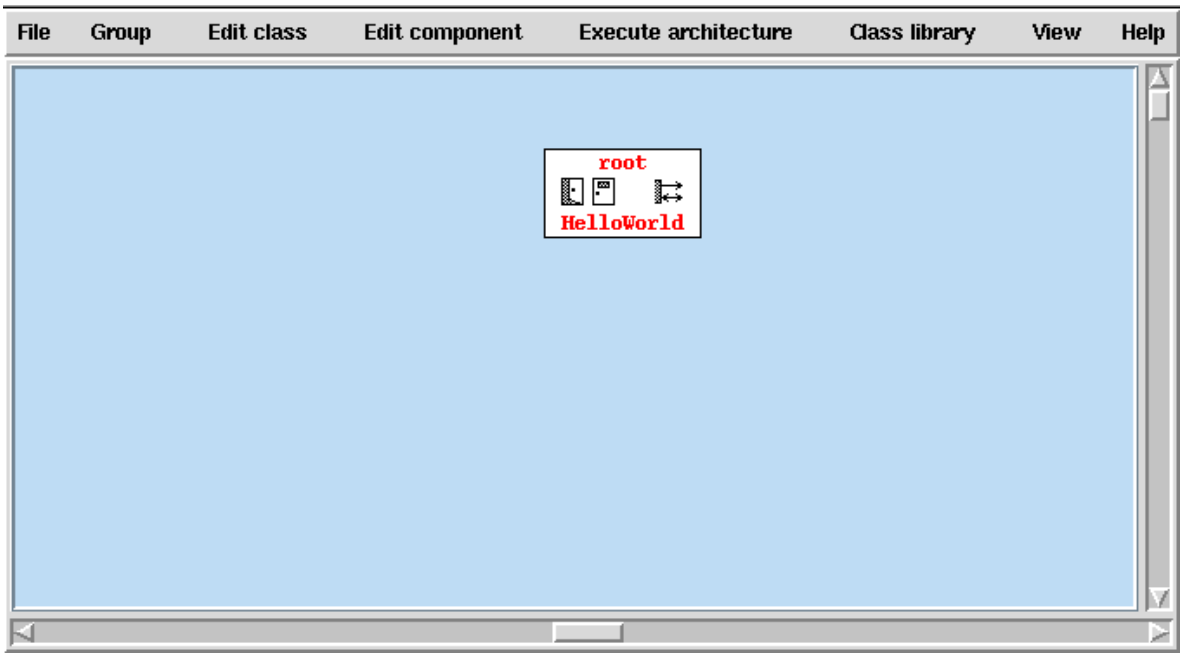
This document assumes the reader has a reasonable familiarity with functional programming in the Haskell language. The details of functional programming are not covered here.

Hello World

To start our discussion of Clock, we begin with the simple example of a program that simply displays "Hello world".

Programs in Clock are developed visually, using the ClockWorks programming environment. This environment is invoked under Unix with the command `cw`. ClockWorks contains facilities for creating, browsing and editing Clock programs, for executing programs, and for creating and accessing libraries of predefined program components.

Programs consist of a set of components organized in a tree structure. The "hello world" program consists of one component:

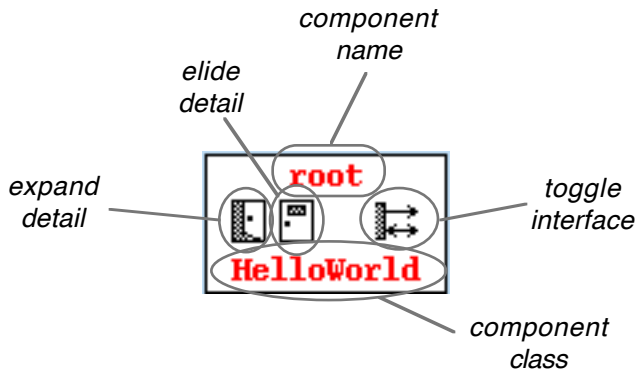


The component itself is programmed textually, in a functional language similar to Haskell. The component consists of the single line:

```
view = Text "Hello world".
```

This states that when this program is executed, a window is to appear containing the text "Hello world".

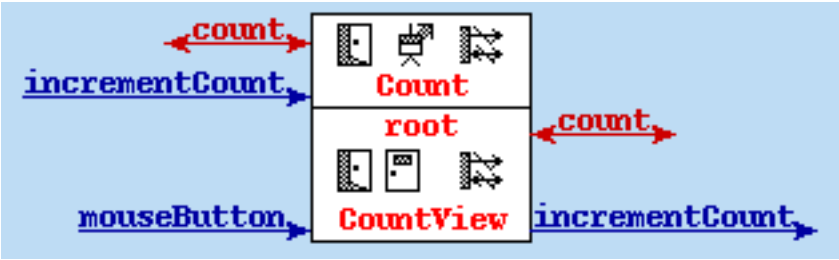
In ClockWorks, components are shown as follows:



The name on the top of the component is the component's name; the name on the bottom is the component's class. Three icons are available for manipulating the appearance of the component: the open and close door icons expand and elide detail; the toggle component is used to show or remove the component's interface. Through ClockWorks facilities for expanding and eliding detail, it is possible to work with complex architectures on displays of normal size.

An Interactive Application

As an example of how to write interactive applications in Clock, the following program implements a counter. When the user clicks on the counter, it is incremented, and the new counter value is displayed. The architecture for this program is:



Consider first the `Count` component. `Count` is an abstract data type (or ADT) that implements a counter. `Count` implements two methods: `count` returns the value of the counter, and `incrementCount` increments the counter. `count` is referred to as a *request* method, since it returns a value. `incrementCount` is an *update* method, since it modifies the state of the ADT.

The `root` component, of class `CountView`, is responsible for implementing the interactive behaviour of the program. This component responds to `mouseButton` input (i.e., the mouse clicks the user will make), and uses the `count` and `incrementCount` methods implemented in Count. Note therefore that arrows drawn on the left of a component show the methods the component implements, while arrows drawn on the right show the methods a component uses.

The complete code for the `CountView` class is:

```
mouseButton "Down" = incrementCount.
mouseButton "Up" = noUpdate.

view = NumText count.
```

The `view` function specifies that this component is to display the current value of the counter. The `mouseButton` function specifies that if the user clicks the mouse button down over the view of this component, the `incrementCount` method is to be invoked. When the mouse button is released, there is no effect (i.e., `noUpdate` is performed.)

In the view function, the current value of the counter is to be displayed as numeric text. This function is a form of constraint – that is, whenever the counter value changes, the view is automatically recomputed to display the correct value. The programmer does not need to manually invoke the view function to update the display when the counter changes value.

The complete code for the `Count` ADT is:

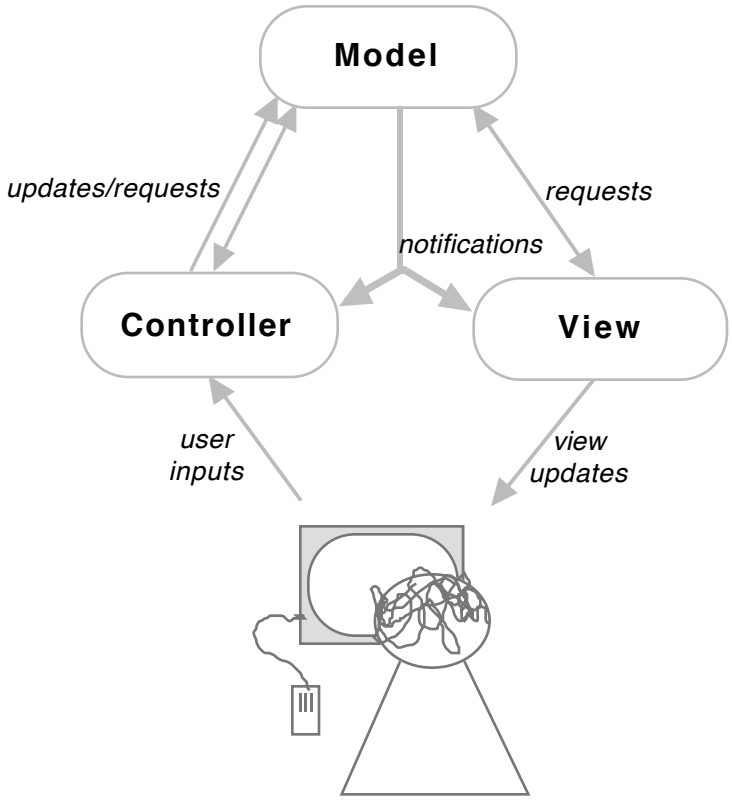
```
incrementCount = save (this + 1).
count = this.

initially = save 0.
```

In this code, the predefined function `this` refers to the current state of the ADT (i.e., the current numeric value of the counter.) The predefined function `save` assigns its argument as the new state of the ADT. Therefore, the `count` method simply returns the current value of the component (`this`). The `incrementCount` method simply saves the value of the current counter value, plus one (`save (this + 1)`).

The `initially` function is invoked when the ADT is created, setting the initial state to being the number 0.

Programming Model

The example of the counter program shown in the section serves to illustrate Clock's programming model. Clock is based on an extended Model-View-Controller (MVC) paradigm. In MVC, programs are split into three components, as shown below:



The *model* represents the underlying state of the application. The *view* encodes how the user interface is to appear on the display. The controller specifies how *input* is to be handled. The key to MVC's power is how these components communicate. User inputs are given to the controller. The controller then determines how these inputs are to be reflected in terms of modifications to the model. The model then announces to the view that the model has changed. The view then updates itself, resulting in a new display. This approach leads to a strong separation of concerns between the model, view and controller. The controller does not directly communicate with the view, only with the model. The view is responsible only for updating the display, and does not need to know the details of when display updates may arise.

To illustrate how MVC is realized in Clock, consider the counter example from the last section:

Model

```
incrementCount = save (this + 1).
count = this.

initially = save 0.
```

Controller

```
mouseButton "Down" = incrementCount.
mouseButton "Up" = noUpdate.
```

View

```
view = NumText count.
```

Here, the `Count` ADT implements the *model*, or the data underlying the application. The `CountView` component implements both the controller and the view. The *controller* consists of a function specifying how mouse button input affects the model – i.e., when the user clicks the mouse button, the counter is incremented. The *view* consists of a function specifying that the current value of the counter is always to appear on the display.

MVC as implemented in Clock therefore gives a clean separation of concerns between data abstraction (the model), input handling (the controller), and display maintenance (the view).

A Multimedia Example

The MVC model also extends to programming multimedia applications. The following program implements playback of a video clip from a file. The output of the program is simply the video as it plays:

The Clock architecture implementing this program is:

Conceptually, a video clip consists of a sequence of frames. Displaying the video therefore involves displaying each frame in turn, at some fixed rate. In the Clock program, the ADT `FrameCounter` is used to represent the number of the frame currently being displayed. This ADT provides a request `frameNumber` to report the number of the current frame, and an update `advanceFrame` to move to the next frame. The complete code of the `FrameCounter` ADT is:

```
frameNumber = this.
advanceFrame = save (this+1).

initially = save 0.
```

The `VideoPlayer` component is responsible for displaying the current video frame and for advancing the frames. As shown in the architecture diagram, the component uses the methods `advanceFrame` and `frameNumber`.

The key issue in displaying video is that the frames must be advanced on a regular

interval – for example, to achieve a frame rate of 10 frames per second, the frame counter must be updated every 100 ms. To trigger updates of the frame counter, the special `tick` input is used. `tick` inputs are issued by the Clock runtime system at some specified interval. By intercepting the `tick` inputs, the `VideoPlayer` component can advance the frame on a regular basis, using the code:

```
tick = advanceFrame.
```

That is, every time the runtime system generates a tick input, the frame counter is advanced.

The view of the video player is:

```
view =
    TickingEvery 100 (
        VideoFrame videoFile frameNumber
    ).
```

Here, the current video frame is: `VideoFrame videoFile frameNumber`. That is, the frame `frameNumber` taken from the video clip contained in the file `videoFile`. Whenever the frame number is incremented, the view function is automatically updated to display the correct frame.

The view function also specifies that the ticking rate of this view is 100 ms. This means that the runtime system will deliver a tick input to this component every 100 ms, leading to a 10 frame per second playback.

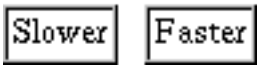
In summary, the complete code for the `VideoPlayer` component is:

```
tick = advanceFrame.

view =
    TickingEvery 100 (
        VideoFrame videoFile frameNumber
    ).

videoFile = homedir + "/ski.mpg".
```

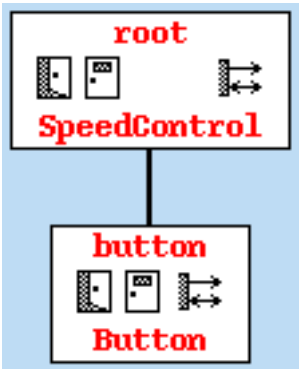
Architectures as Trees

Clock architectures are typically composed of numerous components, organized in a tree structure. Splitting programs into components provides separation of concerns, information hiding, better potential for reuse, and simplifies modification of programs.

To show how architectures are split into trees, imagine we wish to draw two buttons on the display, one labeled with the text "Slower", and the other with the text "Faster". The output should be:



The architecture for this program is:



The `root` component (of class `SpeedControl`) uses the `button` component (of class `Button`) to draw the two buttons. This architecture specifies that the `SpeedControl` component has available a subview called `button` that can be used in composing its own view. The view function of `SpeedControl` is:

```
view =
    beside [
        button "Slower",
        space 10,
        button "Faster"
    ].
```

The view is composed of three parts: a `button` labeled "Slower", a `button` labeled "Faster", and a space of 10 pixels separating them. (The `beside` function composes three views by laying them out horizontally.)

The `Button` component is implemented as:

```
view = groovyBox 1 (pad 2 (Text myId)).
```

That is, the button consists of a grooved box surrounding the textual label of the button, with a space of two pixels around the text. The predefined request `myId` evaluates to the string parameter given to the button by its parent – for example, if this button was created with the function `button "Faster"`, then `myId` would evaluate to the string "Faster".

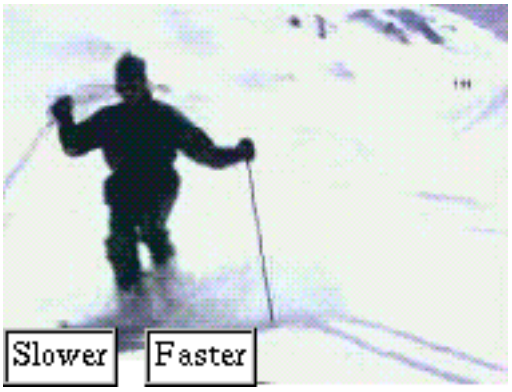
More precisely, when a component has a subview named, e.g., `button`, the component has available a function:

```
button :: String -> DisplayView
```

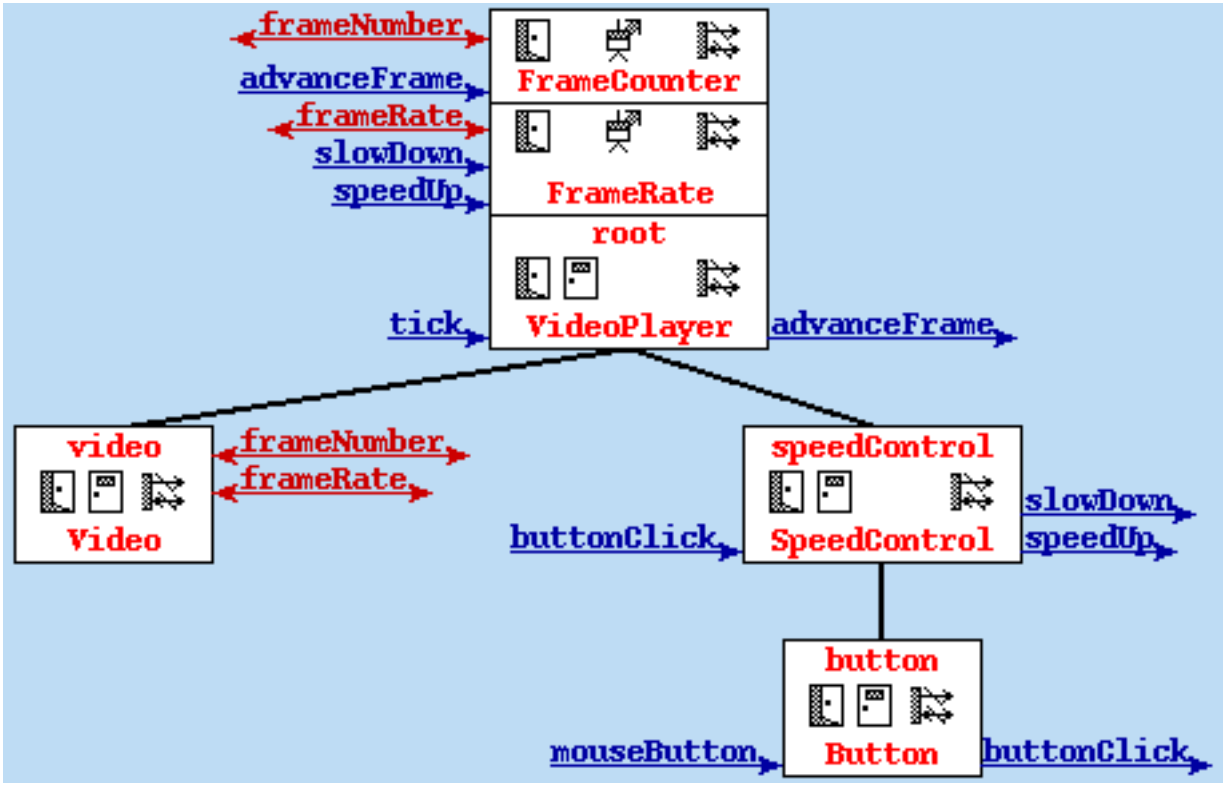
that given a string, returns a view that can be used in creating its own view. In a subview component, this string parameter is accessible via the predefined request `myId`.

A More Complex Example

As a more complex example, we can combine the last two programs, giving a video player with variable playback rate. Whenever the user clicks the "Slower" button, the playback rate slows by 20 ms per frame; the "Faster" button speeds up playback by 20 ms per frame, to a maximum speed of 20 frames per second. The new application looks like:



The architecture implementing this application is:



The revised `VideoPlayer` component composes the views of the current video frame (now implemented in the `video` subview), and the two speed control buttons (implemented in the `speedControl` subview.) The complete code for the `VideoPlayer` is:

```
tick = advanceFrame.

view = Views [video "", speedControl ""].
```

The `Views` constructor allows multiple views to be stacked in 2 1/2 dimensions, so that the speed control appears on top of the current video frame.

Since the rate at which the frame advances is now variable, we add an ADT `FrameRate`, with request `frameRate` to query the frame rate, and updates `speedUp` and `slowDown` to manipulate the rate. The complete code for `FrameRate` is:

```
slowDown = save (this + 20).

speedUp =
    let newRate = max 50 (this-20) in
        save newRate
    end let.

frameRate = this.

initially = save 100.
```

That is, the frame rate is initially 1 frame / 100 ms. Slowing down the frame rate adds 20 ms to this time. Speeding up the frame rate reduces the time by 20, to a maximum speed of 1 frame / 50 ms.

The `Video` component is responsible for displaying the current video frame. The view now specifies that the ticking rate is the current frame rate. The complete code for `Video` is:

```
view =
    TickingEvery frameRate (
        VideoFrame videoFile frameNumber
    ).

videoFile = homedir + "/ski.mpg".
```

The `Button` component is extended to issue a `buttonClick` update when it is clicked. This update carries a parameter of the component's id, to allow parent components to identify which button was clicked. The `Button` code is:

```
mouseButton "Down" = buttonClick myId.
mouseButton _ = noUpdate.

view = groovyBox 1 (pad 2 (Text myId)).
```

`buttonClick` updates are handled by the `SpeedControl` component, which interprets them by adjusting the frame rate. The `SpeedControl` code is:

```
buttonClick "Faster" = speedUp.
buttonClick "Slower" = slowDown.

view = beside [button "Slower", space 10, button "Faster"].
```

That is, if the "Faster" button is clicked, the `speedUp` update is issued to increase the frame rate. Similarly, the "Slower" button results in the frame rate being decreased through the `slowDown` update.

Therefore, applications with sophisticated functionality can be built by composing simple components. The ClockWorks environment allows the high-level structure (or architecture) of programs to be viewed and manipulated visually.

Visibility and Message Routing

Clock's scoping rules state that a component may use any request or update methods appearing above the component in the architecture tree. Conceptually, when a component issues a request or update, the request/update message travels up the tree to the first component with a method by that name. For example, in the video player with speed control of the last section, when the `Button` component issues a `buttonClick` update, the update travels up the tree to the `SpeedControl` component. When the `Video` component issues a `frameNumber` request, the request travels up the tree to the `FrameCounter` ADT. This is a traditional delegation routing scheme.

A consequence of this basic rule that messages travel up the tree to the first ancestor that handles them is that components may only communicate with their ancestors. For example, it is not possible for a component to send a message to its children or siblings.

The reason it is possible to have such restrictive scoping rules in Clock is the presence of constraints used to update views. For example, when the `VideoPlayer` receives a `tick` input updating the frame counter, it is not possible for it to directly inform the `Video` component that a new frame must be displayed – `Video` is a child of `VideoPlayer`. Instead, `VideoPlayer` updates the state of the `FrameCounter` ADT; `Video` depends on the value of the `frameNumber` request, and is automatically updated in response to the change.

Unimplemented Requests and Updates:

If a component issues an update that is not implemented by one of its ancestors, the update is ignored.

If a component issues a request that is not implemented by one of its ancestors, the program terminates with an error message. Some requests, however, have default values. If an unimplemented request has a default value, that value is substituted as the value of the request.

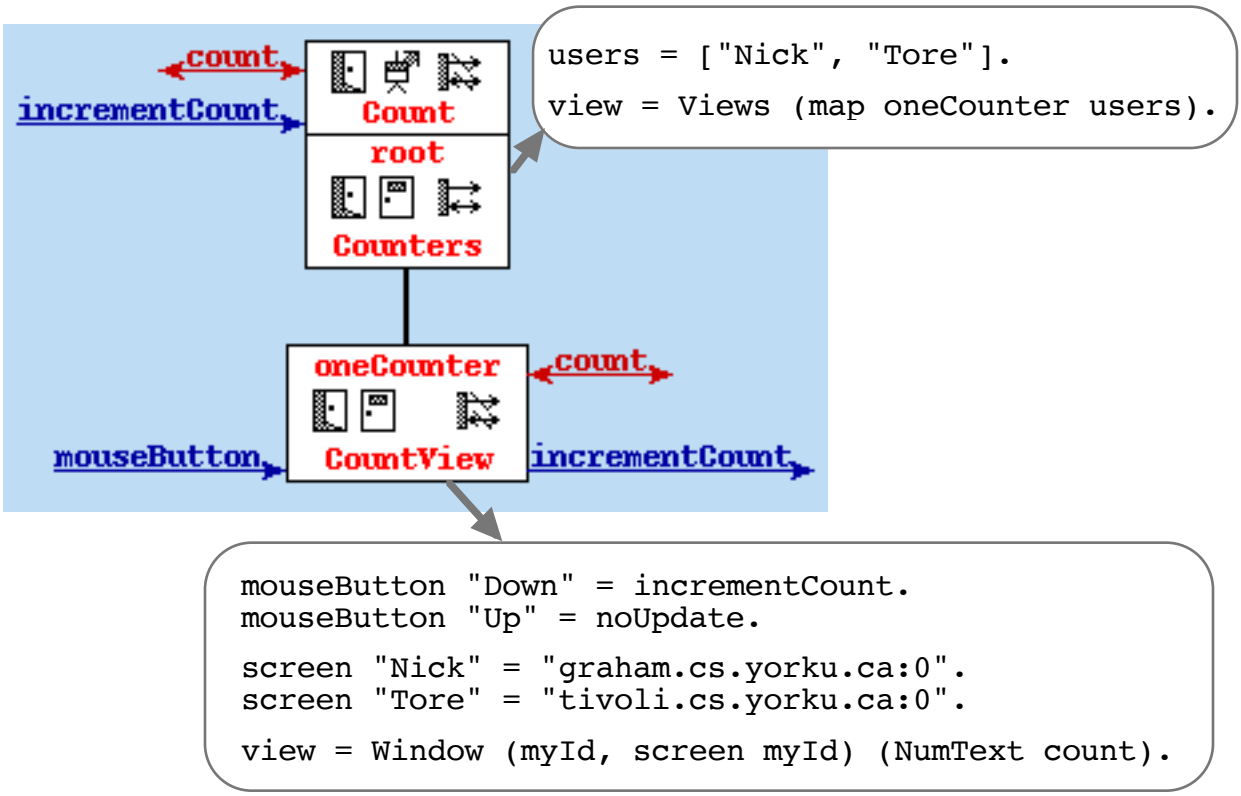
These rules greatly aid in developing programs in an incremental manner, allowing programs to be tested before they are completely implemented. Additionally, library components may be parameterized through requests with default values, allowing them to be more easily integrated into programs.

Multiuser Programs

Clock provides high-level support for creating multiuser programs, where users can share views or customized versions of the same views. Clock automatically handles issues of concurrency and distribution so that programmers do not have to worry about networking or dealing with race conditions.

As a simple example, consider a multiuser version of our counter program. Here, two users (called "Nick" and "Tore") will each have a window displaying a number. When either user clicks on the number, it is incremented on both user's displays.

The code for this application is shown below:



The `Counters` component creates two instances of the counter (`oneCounter`), one for each user (`"Nick"` and `"Tore"`.)

The view of each `CountView` component consists of a window containing the current counter value. The Window constructor has the type signature:

```
Window :: (String,String) -> DisplayView -> DisplayView
```

representing the form:

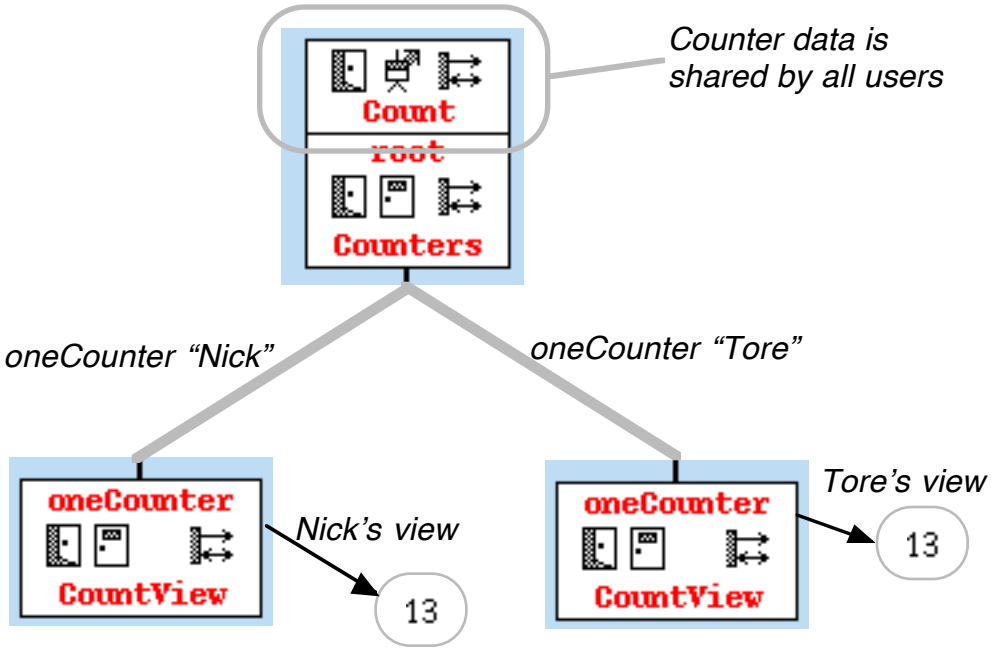
```
Window (windowText, screen) windowContents
```

The *windowText* appears in the title bar of the window. The *screen* is a string containing the IP address of the screen on which the window is to appear. For example, the string `"tivoli.cs.yorku.ca:0"` represents the console of the machine tivoli located at York University. If the screen is specified as the null string (`""`), the window is displayed on the screen represented in the `DISPLAY` environment variable.

The *windowContents* is a `DisplayView` describing what is to appear in the window.

In the counter example, one window will appear on the machine `tivoli.cs.yorku.ca` with the title `"Tore"`, and the other will appear on the machine `graham.cs.yorku.ca` with the title `"Nick"`.

The following diagram demonstrates how this multiuser program works:



When the program executes, the `Counters` component creates two instances of its `oneCounter` subview, one for `"Nick"` and one for `"Tore"`. Each component contains the same view function:

```
view = Window (myId, screen myId) (NumText count).
```

This function specifies that the display is to consist of a window containing the current `count` value, as represented in the `Count` ADT. Therefore, both users' displays show the current count value. If either user clicks on his own counter, the

value in `Count` is modified, triggering the update of both users' displays.

The key points in the design of this multiuser application are:

- Each user's view is represented as a subview of the `Counters` component; i.e., an instance of the `CountView` component.
- Since each user's view is an instance of the same `CountView` component, each user sees the same view on his/her display.
- Each user's view is presented in a different window. The `Window` command has a *screen* parameter that allows the specification of where the user's display is located.
- User's views share data in their ancestor component. I.e., the data in the `Count` ADT is shared by all users. This allows all users' views to be automatically updated whenever one user modifies the count.

This means that the view of one user does not have to be aware of how many other users there are, or where those users are located. The constraint mechanism of having each user depend on shared data allows the views of multiple users to be synchronized automatically.

Multiuser Programs and Session Management

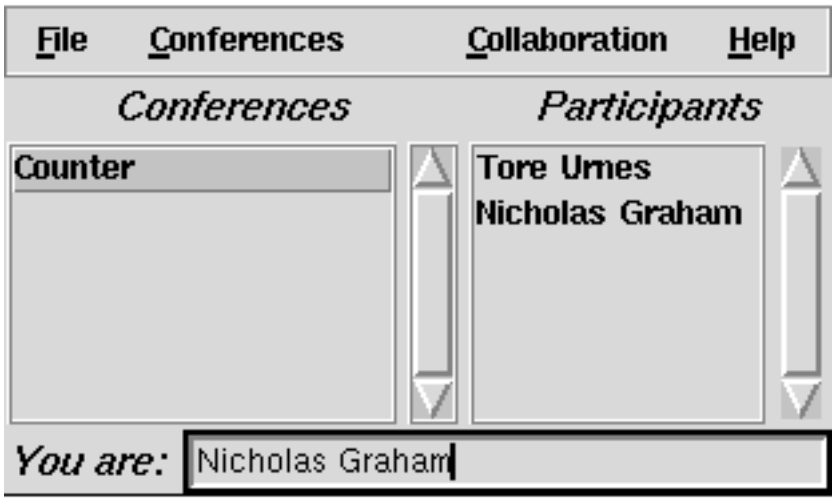
One annoying feature of the last program is that the information about users is hard-coded. That is, the program hard-codes:

- How many users there are.
- What machines those users are working on.

If we ever wish to work on a different machine or add a new user, we have to modify the program.

Clock provides a tool called a session manager[1] that allows users to dynamically enter and leave a *session* (or program.) The session manager allows us to write multiuser programs in a flexible manner, where we do not need to hard code the number of users, the users' names or the users' machines.

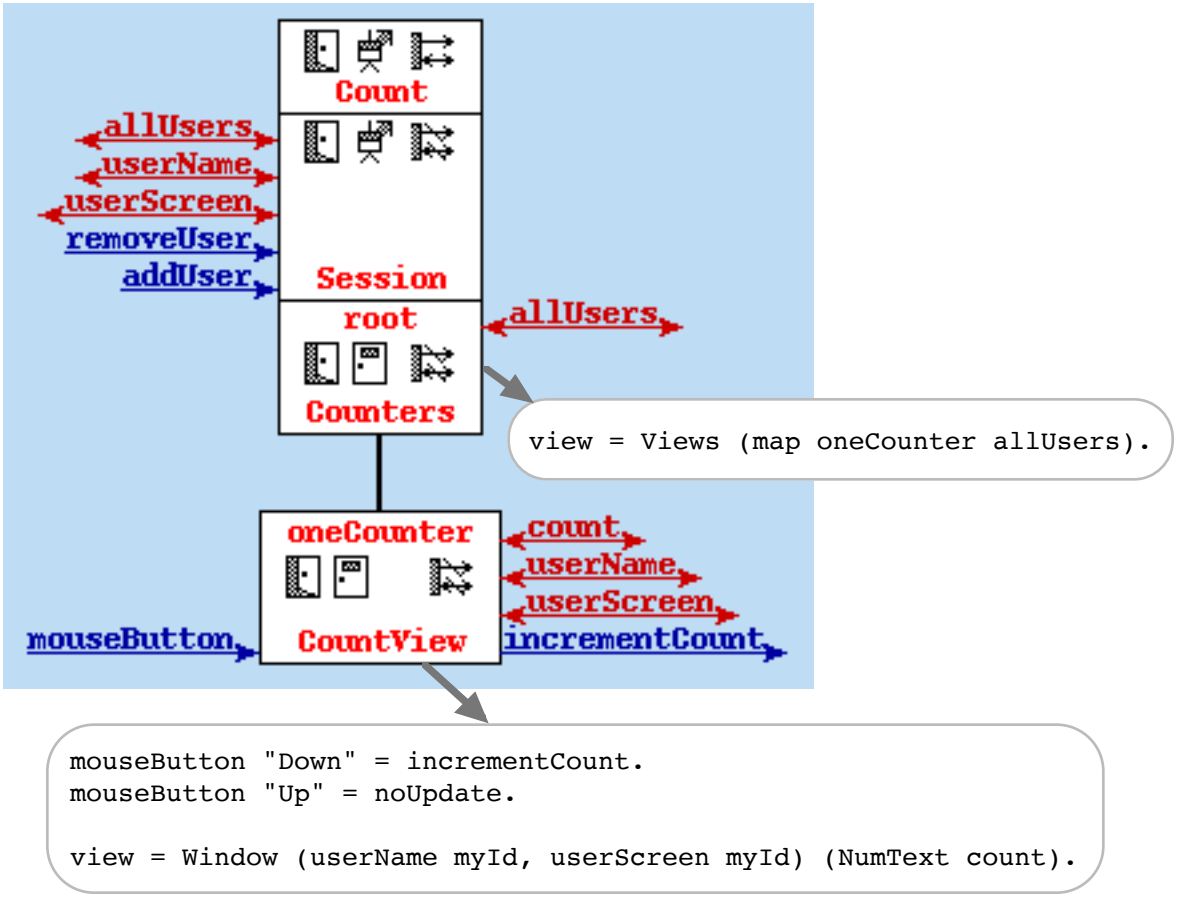
When a user runs the session manager, he/she will see a window that looks like:



This window lists all ongoing conferences – for example, the *Counter* conference refers to the counter program of the last section. When the user clicks on the name of a conference, he/she sees all people currently taking part (i.e., all the current users in the *Counter* program.) When the user double-clicks on the name of a conference, he/she joins it.

Clock programs can be modified to take advantage of the session manager by including the `Session` ADT, as shown below:

[1] The GroupKit session manager by Roseman and Greenberg at the University of Calgary



The `Session` ADT implements the updates `addUser` and `removeUser`. These are invoked by the session manager as users enter or leave the session. Since the session manager keeps track of who is in the session, Clock programmers never need to deal with adding and removing users. The ADT also implements the following requests that help in managing session information:

`allUsers :: [String]`

Gives a list of all current participants in the session. Each participant is assigned a unique string id by the session manager. `allUsers` returns the unique id's of all current participants as a list of strings. `allUsers` is used in the `Counters` component to create one view per user.

`userName :: String -> String`

Given the unique id of a participant, returns the participant's name. In this example, `userName` is used in `CountView` to retrieve the name of the participant so that it can be included in the title bar of the participant's window.

`userScreen :: String -> String`

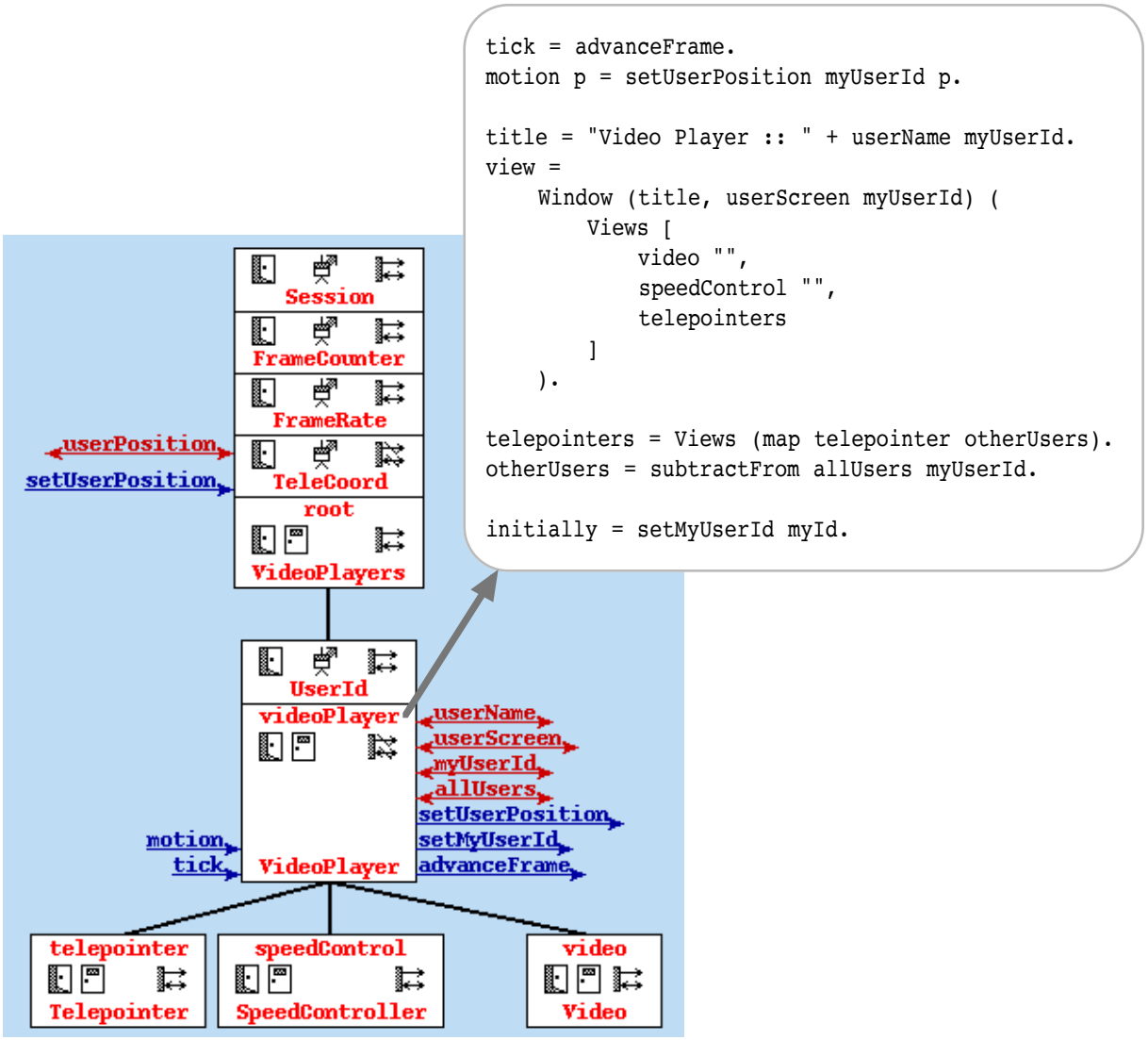
Given the unique id of a participant, returns the participant's screen. In this example, `userScreen` is used in `CountView` to retrieve the name of the participant so that the window can be displayed on the participants display.

The code implementing the multiuser counter makes use of these requests, allowing the application to be coded without hard-coding the number of participants or the location of their displays.

Relaxed WYSIWIS

The counter program of the last section is an example of a *pure WYSIWIS* program. WYSIWIS stands for "What you see is what I see", meaning that the views of all users are completely synchronized.

To appear...



Context Free Syntax of Clock

```
componentDef    ::=   { definition }

definition      ::=   equation
                  |   typeDefinition
                  |   dataDefinition

equation        ::=   id { simplePattern }+ = expn .

typeDefinition  ::=   type typeId = typeSpec .

dataDefinition  ::=   data constructorId =
                        constructorId { typeSpec }
                      { | constructorId { typeSpec }} .

expn            ::=   simpleExpn
                  |   simpleExpn simpleExpn
                  |   debug simpleExpn

simpleExpn      ::=   id
                  |   literal
                  |   constructorId
                  |   fn { simplePattern }+ -> expn end fn
                  |   if expn then expn
                      { elsif expn then expn }
                      else expn
                      end if
                  |   case expn of
                        { pattern -> expn }+
                      end case
                  |   let { pattern = expn }+ in expn end let
                  |   [ expn { , expn } ]
                  |   unaryOp simpleExpn
                  |   simpleExpn binaryOp simpleExpn
                  |   ( expn )
```

simplePattern **::=** **_**
| *id*
| *literal*
| *constructorId*
| **[** *pattern* { **,** *pattern* } **]**
| **(** *pattern* **)**

pattern **::=** *simplePattern*
| *constructorId* { *simplePattern* }
| *simpleExpn* **:** *simpleExpn*

unaryOp **::=** **-**

binaryOp **::=** **+** | **-** | ***** | **div** | **mod**
| **=** | **~=** | **<** | **>** | **<=** | **>=**
| **and** | **or** | **not**
| **@** | **#** | **++**

typeSpec **::=** **Num** | **String** | **Bool**
| *typeId*
| **[** *typeSpec* **]**
| **(** *typeSpec* { **,** *typeSpec* } **)**

Lexical Rules

numLiteral **::=** `[0-9]*`

stringLiteral **::=** `" [any char]* "`

boolLiteral **::=** **True** | **False**

id **::=** `[a-z]* [a-zA-Z0-9_]* ['']*`

constructorId **::=** `[A-Z]* [a-zA-Z0-9_]* ['']*`

typeId **::=** `[A-Z]* [a-zA-Z0-9_]* ['']*`

Predefined Functions

List manipulation functions:

```
take :: Num -> [a] -> [a]
```

Takes the first n elements of the given list; ie, `take 3 [1,2,3,4] = [1,2,3]`.

: :: a -> [a] -> [a]

The 'cons' function places a given element onto the front of a list.

```
append :: [a] -> a -> [a]
```

Appends the given value to the end of the given list.

```
++ :: [a] -> [a] -> [a]
```

Concatenates two lists.

```
assign :: [a] -> Num -> a -> [a]
```

Gives a list where the element at the given position is replaced with the new given element. Ie, `assign [10,11,12,13] 2 1001 = [10,1001,12,13]`.

```
@ :: [a] -> Num -> a
```

Returns the value at the given position of the list. Ie, `[10,9,8] @ 2 = 9`. List indeces are based at 1. List indeces out of bounds result in an error.

```
# :: [a] -> (Num,Num) -> [a]
```

Returns all elements in the given list between the two given numbers. Ie, `[1,2,3,4,5,6] # (2,5) = [2,3,4,5]`. If the right index exceeds the left index by 1, the null list is returned: ie, `[1,2,3] # (1,0) = []`. If either index is otherwise out of bounds, an error results.

```
length :: [a] -> Num
```

Returns the number of elements in the given list.

Dictionary functions:

A dictionary is a list of key/value pairs. The dictionary functions make it easy to retrieve values from a dictionary given a key.

```
enterDict :: a -> b -> [(a,b)] -> [(a,b)]
```

Given a key *a*, a value b, and a dictionary, returns a new dictionary where b has been entered under key a. If the key a already exists in the given dictionary, this operation overwrites the old value.

```
lookup :: a -> [(a,b)] -> b
```

Given a key *a*, finds the first value b with that key in the dictionary.

```
lookupPos a -> [(a,b)] -> Num
```

Given a key a, finds the position of the first occurence of that key in the dictionary.

```
lookupAll :: a -> [(a,b)] -> [b]
```

Given a key a, returns the list of all values b with that key in the dictionary.

```
index a -> [a] -> Num
```

In a simple list, finds the position of the first occurence of a value in that list. Ie, `index 3 [1,3,10,4] = 2`.

Combinators:

```
map :: (a->b) -> [a] -> [b]
```

Given a function *f* and a list of elements *as*, gives the result of applying *f* to each element of *as*.

```
fold :: (a->a->a) -> a -> [a] -> a
```

Given a function f, initial value a0, and list of elements [a1,a2,...,an], returns the result of applying f a0 (f a1 (f a2 (..... (f an-1 an))).

Mathematical functions:

```
min :: Num -> Num -> Num
max :: Num -> Num -> Num
```

Return the min/max of the given pair of numbers.

Tuple functions:

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

Return the first/second items of a tuple.

String manipulation functions:

```
ord :: String -> Num
```

Returns the ordinal number of the first character of the given string; eg, in ASCII, `ord ("A") = 65`. The given string must not be null.

```
chr :: Num -> String
```

Gives as a string the character value of the given integer. Eg, `ord (65) = "A"`.

```
numstr :: Num -> String
```

Converts an integer into a string. Eg, `numstr 123 = "123"`.

```
strnum :: String -> Num
```

Converts a string into an integer. Eg, `strnum "123" = 123`. The string must contain a valid integer; currently this is not checked, so if the string is not valid, a segmentation violation style of crash will result.

Date/time manipulation functions:

Date and Time are predefined abstract data types representing a date/time. These functions implement computations based on dates. Note that in Clock, there is currently no way of accessing the current date or time.

```
textToDate :: String -> String -> String -> Date
```

Creates a date type from the given day, month and year. The month must be the full English name of the month: "January", "February", etc. For example, `textToDate "10" "January" "1995"` gives the date for Jan 10, 1995.

```
dateToString :: Date -> String
```

Gives a string representation of a date.

```
dayOfWeek :: Date -> String
```

Returns what the weekday is of the current date.

```
daysInMonth :: Date -> Num
```

Returns how many days are in the current month.

```
daysInYear :: Date -> Num
```

Returns how many days are in the given year. Does a reasonable job with leap years, but not perfect.

```
tomorrow :: Date -> Date
```

Gives the date following the given date.

```
yesterday :: Date -> Date
```

Gives the date preceding the given date.

```
stringToTime :: String -> Time
```

Converts a string to a time. Time strings can be in two formats: 24 hour, such as `stringToTime "13:45"` or 12 hour, such as `stringToTime "1:45 PM"`. This function is very sensitive to format, and will crash if there is any deviation.

```
timeToString :: Time -> Num -> String
```

Returns a string version of the given time. The time may be in 12 or 24 hour format, as specified by the numeric parameter.

```
earlier :: (Date,Time) -> (Date,Time) -> Boolean
```

Determines whether the first of two given dates+times occurs earlier than the second.

Environment query functions:

```
getenv :: String -> String
```

Given the string name of an environment variable, returns the variable's value. Eg, `getenv "PRINTER"` returns the name of the current printer.

```
homedir :: String
```

Returns the directory in which this program is located. This function is useful if the program makes use of auxiliary files located in the program directory (e.g., image files.)

Views in Clock

Clock provides a simple language from which graphical displays can be constructed. While the language is simple, the abstraction powers of functional programming can allow complex displays to be easily constructed. Currently, the view language does not support all forms of graphical primitives or layout that would be desirable; the most glaring problem right now is the lack of any circle or elipse primitives.

This presentation begins by outlining the primitive constructs from which views are built, and then describes the predefined functions that simplify view manipulation.

Primitive Constructs

Views are elements of the data type `DisplayView`. This data type has the following definition:

```
data DisplayView =
    Views [DisplayView]
  | Line Coord Coord
  | Arrow Coord Coord
  | At Coord Coord DisplayView
  | Box DisplayView
  | Text String
  | NumText Num
  | CharText Char
  | BooleanText Bool
  | InstanceOf SubViewName SubViewId

  | BorderStyle Num DisplayView
  | BorderColour Colour DisplayView
  | BorderWidth Num DisplayView
  | FillPattern Num DisplayView
  | FillColour Colour DisplayView
  | Font FontName DisplayView
  | FontColour Colour DisplayView
  | Inverted DisplayView
  | LineWidth Num DisplayView
  | LineStyle Num DisplayView
  | LineColour Colour DisplayView
  | SaveProps DisplayView
  | RestoreProps DisplayView

  | DarkReliefShade Colour DisplayView
  | LightReliefShade Colour DisplayView
  | ReliefWidth Num DisplayView
  | Relief String DisplayView
  | PolyLine [Coord]
  | Pile [(Num, DisplayView)]
  | Image String
  | Crop Coord Coord DisplayView.
```

Views in Clock are therefore values that are built from data constructors. For example, the view `Box (Text "Hello world")` represents the text "Hello world" drawn with a box around it.

Text, NumText, CharText, BooleanText

These constructors are used to display text. `Text "Hello"` displays the text "Hello". `NumText 123` displays the text "123". `CharText 65` displays the text "A". `BooleanText True` displays the text "True". The text is drawn in the current font and current font colour. For example, `Font largeItalicFont (FontColour red (Text "Hello world"))` draws the text "Hello world" in large, red, italic text.

Fonts are strings in the standard X font description format. The Unix command `xlsfonts` will give you the complete list of X fonts available on your server. The file `$CLOCKSYS/Source/clocklib/viewUtils` gives a list of predefined font names. Note that your server may not support all the fonts defined in this file.

Colours have the definition:

```
type Colour = (Num,Num,Num).
```

That is, colours are triples consisting of an integer value representing the red, green and blue intensities of the desired colour. The file `$CLOCKSYS/Source/clocklib/colourUtils` gives a list of several hundred predefined colour names.

Box

A box surrounds whatever its parameter view is. E.g., `Box (Text "Hello")` draws a box correctly sized to fit the text "Hello". By default, boxes are drawn in black, with a width of one pixel. The constructors `BorderStyle`, `BorderColour`, `BorderWidth`, `FillPattern` and `FillColour` permit the attributes of boxes to be adjusted. For example,

```
BorderWidth 3 (
  FillColour yellow (
    BorderColour green (
      Box (Text "Hello")
    )
  )
)
```

displays the text "Hello" surrounded by a 3 pixel wide green border, on a yellow background. Border styles may be: `solidBorder` (default), `dashedBorder`, or `doubleDashedBorder`. Fill patterns may be: `solidFill` (default), `hashedFill`, `screenDoorFill`, or `tiledFill`.

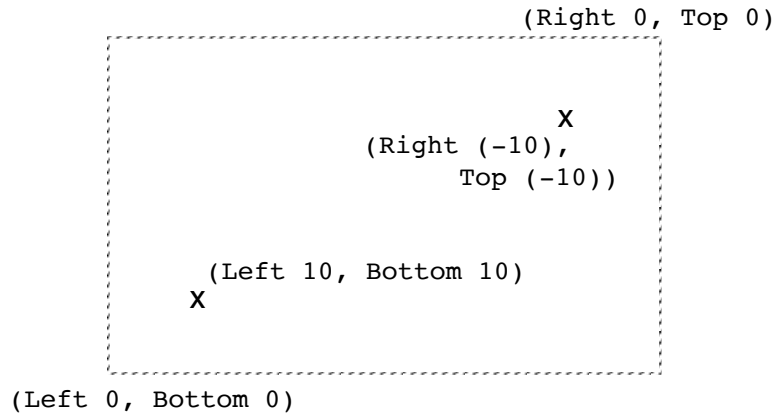
At

To place Clock primitives on the display, coordinates must be used. Coordinates refer to positions within the current *canvas*. The contents of each box construct is considered to be a separate canvas with its own coordinate space, thus giving Clock a hierarchical graphics system. The *Coord* data type defines the form of coordinates:

```
type Coord = (Ordinate,Ordinate).
type Offset = Num.
type OrdinateLabel = Num.
```

```
data Ordinate =
    XBaseOffset OrdinateLabel Offset | YBaseOffset OrdinateLabel Offset
  | Left Offset  | Bottom Offset | Right Offset | Top Offset
  | XSomewhere | YSomewhere.
```

The simplest form of coordinate is the absolute coordinate. The form `(Left 0, Bottom 0)` is used to specify the position 0 units to the left of the lower-left corner of the coordinate space. Similarly, the form `(Right 0, Top 0)` specifies the upper-right corner of the coordinate space. This form of absolute coordinate allows primitives to be located without knowing the position or size of the coordinate space itself:



Primitives can be positioned with the `At` constructor. For example:

```
At (Left 10, Bottom 10) (Right (-10), Top (-10)) (Box noView)
```

would display a box whose size would be adjusted to always be 10 pixels from the border of the canvas.

As a convenience, the functions:

```
x xpos = Left x.
y ypos = Bottom y.
origin = (x 0, y 0).
```

are predefined. These functions allow simpler forms for coordinates that are expressed in terms of the lower-left corner of the canvas.

Sometimes, the programmer does not know the position of both coordinates for a primitive. For example, when positioning text, it is possible to write:

```
At (x 20, y 13) (XSomewhere, YSomewhere) (Text “Hello”)
```

This states simply that the lower-left corner of the text is to be positioned at (20, 13)

within the current coordinate space, and the upper-right corner's position is not specified.

As a convenience, the function:

```
stretching = (XSomewhere, YSomewhere).
```

is defined. With this form, text can be simply positioned as:

```
At (x 20, y 13) stretching (Text "Hello")
```

Views

The views constructor allows multiple primitives to be placed in the same canvas. For example,

```
Views [
   At (x 10, y 10) stretching (Text "Hello"),
   At (x 100, y 10) stretching (Text "there")
]
```

places two text primitives in the same canvas.

In specifying the positions of primitives within a `Views` construct, primitive coordinates can also be expressed relative to the positions of other primitives. For example, to place two texts on the display separated by 10 pixels, one writes:

```
Views [
   At (x 10, y 10) (XBaseOffset 1 0, YSomewhere) (Text "Hello"),
   At (XBaseOffset 1 10, y 10) stretching (Text "there")
]
```

The `XBaseOffset` constructor takes two parameters: the first is a numeric name for a position on the display. In this case, the name "1" is used to refer to the X-position of the right extent of the text "Hello", wherever that may be. The second parameter is an offset (positive or negative) from that position. Therefore, the text "there" is positioned at `XBaseOffset 1 10`, which is 10 pixels to the right of the rightmost position of the text "Hello".

Line, Arrow, PolyLine

These constructors allow lines and arrows to be drawn. For example,

```
Line origin (x 10, y 10)
```

is a line stretching from the origin to the position (10, 10) in the current canvas.

The constructors `LineWidth`, `LineStyle` and `LineColour` allow the attributes of lines to be set. Currently, arrows don't have arrow heads.

PolyLine allows polygons to be built from a list of coordinates. The current fill colour is applied to the region contained by the polyline.

Arrow may have unpredictable effects in the current version of Clock.

SaveProps, RestoreProps

`SaveProps` saves the current state of all properties (line colour, font, fill colour, etc). `RestoreProps` restores the last saved set of properties. For example, to define a function to draw a green box around a given view, we would write:

```
greenBox v = SaveProps (BorderColour green (RestoreProps v)).
```

Ie, we save the properties prior to changing the border colour, and then restore them before evaluating `v`.

Image

The `Image` constructor allows the inclusion of JPEG images in Clock programs. For example, `Image "foo.jpg"` reads the image contained in the file "foo.jpg". Images are first class Clock views; for example, `Box (Image "foo.jpg")` draws a box surrounding the image.

Crop

Crop clips the given view to fit within the given coordinates. For example,

```
Crop (x 10, y 10) (x 30, y 30) (Image "foo.jpg")
```

displays the portion of the image contained in the region (10,10) -> (30,30).

Relief

Relief allows the current view to be given the 3D style of relief commonly used in modern toolkits. The form `Relief "raised" v` draws v slightly raised over the surrounding view; `Relief "sunken" v` draws v slightly sunken. Relief properties include the `ReliefWidth`, the number of pixels wide the relief shading will be, and `DarkReliefShade` and `LightReliefShade`, which are used to specify the colours to be used in relief shading.

Predefined View Functions

The last section introduced the low level primitives from which all Clock views are constructed. In fact, Clock programmers do much of their view construction using predefined functions that abstract from the detailed level of the primitives. These definitions are all contained in the files `$CLOCKSYS/Source/clocklib/viewTypes` and `$CLOCKSYS/Source/clocklib/viewUtils`. Reading through these files is a very useful way of finding out how to write sophisticated view functions in Clock.

Coordinates

```
x :: Num -> Ordinate
y :: Num -> Ordinate
```

Allow the specification of coordinates relative to the lower-left corner of the current canvas. Eg, `(x 10, y 20)` is a coordinate.

```
xOrigin :: Ordinate
yOrigin :: Ordinate
```

Return the lower left X- and Y-positions of the the current canvas.

```
origin :: Coordinate
```

Returns the lower-left coordinate of the current canvas.

```
mostHigh :: Ordinate
mostRight :: Ordinate
topRight :: Coordinate
```

Return the positions of the top-right extent of the current canvas.

```
stretching :: Coordinate
```

Returns a coordinate at an unspecified location.

```
noView :: DisplayView
```

Returns no view at all. Useful in, for example `Box noView`, a box containing nothing.

Layout

```
beside :: [DisplayView] -> DisplayView
above :: [DisplayView] -> DisplayView
```

Given a list of display views, returns the views laid out horizontally/vertically respectively. For example, `beside [Text "a", Text "b", Text "c"]` would display the text "abc".

```
size :: (Num,Num) DisplayView -> DisplayView
```

Makes the given view into the specified size. This is useful, for example, in creating a box of a specific size without having to specify its position: `Size (10, 10) (Box noView)`.

```
group :: DisplayView -> DisplayView
```

Introduces a new canvas around the given display view. This allows a complex view to be included into a new view which may have a conflicting coordinate space.

Shadows

```
shadow :: Num -> DisplayView -> DisplayView
```

Draws a drop-shadow around the lower-left of the given view. The current fill colour is used as the colour of the shadow.

```
whiteShadow :: DisplayView -> DisplayView
blackShadow :: DisplayView -> DisplayView
greyShadow :: DisplayView -> DisplayView
greenShadow :: DisplayView -> DisplayView
```

Draws shadows of the named colour.

```
upperShadow :: Num -> DisplayView -> DisplayView
whiteUpperShadow :: DisplayView -> DisplayView
```

Same idea as above, except the shadows are drawn to the upper-right.

Relief Helper Functions

```
motifShading :: Bool -> Num -> DisplayView -> DisplayView
```

Adds a Motif-style border to the given display view. The numeric parameter specifies the width of the shading. The boolean parameter if True specifies sunken relief, if False specifies raised relief.

```
groovyBox :: Num -> DisplayView -> DisplayView
```

Adds a "groovy" box of two times the specified width around the given display view.

Padding and Spacing

```
pad :: Num -> DisplayView -> DisplayView
```

Adds a blank border of the specified number of pixels around the given display view.

```
paddedText :: Num ->  String -> DisplayView
```

Returns a display view consisting of the string with a blank border of the specified number of pixels surrounding it.

```
space :: Num -> DisplayView
hSpace :: Num -> DisplayView
vSpace :: Num -> DisplayView
```

Returns a space of the given size in pixels. This function is useful for spacing out items listed in a beside or above function. *space* returns a square space; *hSpace* and *vSpace* have height and width of 1 pixel respectively.

```
spaceApart :: [DisplayView] -> [DisplayView]
```

Inserts a two-pixel wide space between each element of the given list of views.

```
largeSpaceApart :: [DisplayView] -> [DisplayView]
```

Inserts a ten-pixel wide space between each element of the given list of views.

Grabbing

By default, input is directed to the component whose view is under the tracking symbol. Ie, clicking input is directed to whatever is clicked. Sometimes, it is desirable to explicitly grab inputs when some condition is met. This grabbing is specified in the view language. For example, the typical implementation for a button is:

```
view =
    if isDepressed then
        GrabbingMouseButton (Box (Text myId))
    else
        Box (Text myId)
    end if.
```

Here, whenever the button is depressed, all next mouse button inputs will be sent to the button until it is released. This means that if the user clicks on a button and then moves the mouse before releasing, the button "up" input will still be sent to the button.

The grabbing directives are:

```
GrabbingMouseButton :: DisplayView -> DisplayView
```

Grabs subsequent mouse button inputs.

```
GrabbingMouseMotion :: DisplayView -> DisplayView
```

Grabs subsequent mouse motion and relative motion inputs.

```
GrabbingMouse :: DisplayView -> DisplayView
```

Grabs subsequent mouse button, motion and relative motion inputs. This directive has the same effect as `GrabbingMouseButton` and `GrabbingMouseMotion` combined.

```
GrabbingKeyboard :: DisplayView -> DisplayView
```

Grabs subsequent keyboard inputs.

The grabbing directives are stack-based: if a component grabs a resource currently belonging to another component, it takes the resource. When the second component releases the resource, its ownership reverts to the first.

User Inputs

Clock supports a number of predefined user inputs to allow access to events generated by the mouse and keyboard input devices. To access these inputs, an update function must be placed in the event handler taking the input.

Mouse Button

By default, mouse button input is directed to the nearest enclosing event handler over which the mouse is clicked.

```
mouseButton :: String -> UpdateEvent
```

Registers that a mouse click has been performed over the view of the event handler taking the `mouseButton` input. The string parameter can be `"Down"`, indicating the mouse button has been depressed, or `"Up"`, indicating the mouse button has been released. On multi-button mice, all buttons generate the `mouseButton` input; there is no way of distinguishing which button was depressed.

Mouse Motion

By default, mouse motion events are directed to the event handler whose view most tightly encloses the postion at which the mouse motion occurs. The granualarity of mouse motion events is dependent on the windowing system; in most systems, moving the mouse rapidly causes motion events to be dropped, possible also leading to `enter` events being missed.

```
enter :: UpdateEvent
```

Indicates the mouse has entered over the view of the event handler taking this update.

```
leave :: UpdateEvent
```

Indicates that mouse was over the view of the event handler taking this update, but now is not.

```
motion (Num,Num) :: UpdateEvent
```

Indicates that mouse is now located at the given coordinate within the view of the event handler taking the update. The coordinate system of this event handler is used, so the coordinate `(0,0)` indicates the lower-left corner of the view.

```
relMotion (Num,Num) :: UpdateEvent
```

Indicates that the mouse has moved by the given number of pixels in the X and Y direction since the last `relMotion` update was given.

Keyboard

Keyboard input is directed to the last event handler to grab keyboard input. If nobody has grabbed the keyboard, inputs are discarded.

```
key :: Num -> UpdateEvent
```

Indicates that a key has been depressed. The parameter represents the ordinal of the key clicked.

```
arrowKey :: String -> UpdateEvent
```

Indicates that one of the arrow keys has been depressed. The string parameter indicates which arrow key, and can have values of `"Left"`, `"Right"`, `"Up"` or `"Down"`.

```
editKey :: String -> UpdateEvent
```

Indicates that a special key has been depressed. The string parameter indicates which key was depressed, and can have values of `"Tab"`, `"Backspace"`, `"Delete"`, `"Escape"` or `"Return"`.

```
functionKey :: String -> UpdateEvent
```

Indicates that a function key has been depressed. The string parameter indicates which function key, and can have values of `"1"` through `"35"`.

Predefined Updates

The following updates are predefined in the language. There are currently no predefined requests.

All or Nothing

```
all :: [UpdateEvent] -> UpdateEvent
```

In order to permit update functions to return more than one update, the `all` update groups a list of updates into a single update. All updates are performed, in no specified order.

```
noUpdate :: UpdateEvent
```

The update noUpdate indicates that no update is to be performed. This is equivalent to `all []`.

Known Problems

Clock is still an experimental system, and has a number of known bugs and shortcomings. These will hopefully be resolved over time.

Layout

There is a known bug in the layout routine that sometimes causes views to be allocated more space than they should. The cases under which this occurs are hard to describe precisely. So far, only I have actually come up with an example that triggers this error.

Type Checking

There is no type checker currently, so most type errors will result in "segmentation violation" or "bus error" types of crashes. Note that in functional languages, type errors include type mismatches on operations, and providing too many or too few parameters to functions or constructors. Tracking down type errors is not actually all that hard – use the trace option in cw to locate the component and function in which the crash occurs, and use the 'debug' function to check that you are indeed getting the values you expect at various points.

2.5 D

Currently, 2.5 D layout is supported through an awkward and inefficient mechanism. This is due for a total overhaul. In the current system, if you try placing objects in a layered manner, you may get unpredictable results.

Output

There is no way in Clock of drawing circles, elipses, or splines.

Reals

Clock currently provides no support for real numbers. The `Num` type is currently integer.

External Interface

There is currently no way of accessing code written in other languages or the environment in general.

Loading Libraries

Currently, when you load a library in ClockWorks, you get access to the class definitions in the library, but not to the actual code of the library components. If you run programs using library components, you will get error messages of the form:

```
Clint (Fatal Error): Cannot open '/cs/u/graham/EClock/Programs/temp/Depressed'
```

To create links to the code of the library components, you must:

- cd to the project directory
- perform the command: `cplib` *libraryName*

For example, if the project is called 'MyProject', and the library name is 'Buttons', then type:

- `cd $CLOCKSYS/Programs/MyProject`
- `cplib Buttons`

It is often convenient to include `$CLOCKSYS/Programs` in your cdpath.