Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Interaction objects in the made multimedia environment

F.C. Heeman, I. Herman and G.J.P. Reynolds

Computer Science/Department of Interactive Systems

CS-R9516 1995

Report CS-R9516
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Interaction Objects in the MADE Multimedia Environment

Frans C. Heeman, Ivan Herman, Graham Reynolds

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email {frans,ivan,reynolds}@cwi.nl

Abstract

In order to support flexible interaction in interactive multimedia systems, we have defined general interaction objects that allow dynamic and transparent coupling of user interaction to media objects and other types of objects. The interaction objects are programmable in that they may use scripting languages for defining their behavior. This mechanism can be used to glue existing application capabilities together to form new end-user functionality.

CR Subject Classification (1991): D.1.3, D.1.5, D.3.3, D.3.4, H.5.1, H.5.2, I.3.6
Keywords & Phrases: MADE, multimedia, user-interaction architecture, scripting, finite state machines.
Note: This paper has been published in the Proceedings of the "Eurographics Symposium on Multimedia/Hyper media in Open Distributed Environments" (Graz, Austria), by Springer-Verlag, pp. 264-277, June 1994.

1. Introduction

The definition of user interaction is often one of the most difficult tasks to perform during the development of an interactive system. Usually an iterative approach is taken, so that different interactions can be tested out and refined. This is best achieved when a flexible means to define and change interaction is provided, independently from an application's functionality. Multimedia systems in particular, are often more complex on the application side because of their need to handle multiple, time-critical streams of high volume distributed data, and consequently they may push aside the user interface as a secondary issue. Having a means to separate the interaction from the application side makes it possible to give the development of the user interface its proper place.

In the European Communities' ESPRIT III project MADE (Multimedia Application Development Environment [3, 7]), a portable object-oriented development environment is being defined and implemented for multimedia applications. The outcome of this work will be a programming environment, based on C++, running on various UNIX platforms and on Windows-NT environments.

A flexible scheme has been defined as part of the MADE project for handling user interaction. Essentially, it consists of general interaction objects based on finite state machines, where the behavior of the state machine can be defined either as part of the interaction object itself or separately in a script. State transitions are triggered by events originating from input devices or from other objects. General software interfaces have been defined, so that interaction objects defined in MADE programs can access scripting languages in a way that is independent of the specific scripting language used, and conversely, scripting languages can invoke application functionality defined in a MADE program.

Section 2 of this paper describes the general considerations that underly our design. The actual architecture is given in Section 3. Some examples in Section 4 show the concrete application of the design to typical multimedia scenario's. Conclusions are given in Section 5.

2. Design Considerations

As discussed in the introduction, we think it is important to be able to separate the definition of user interaction from the actual application functionality. In the design of interactive systems, this requirement comes down to separating the definition of interaction from the objects being manipulated. Furthermore, to be able to iterate quickly over changes in the user interface it must also be possible to change the definition of interaction dynamically, without touching the application in any way. Taking these two points together, we arrive at the notion of *programmable interaction objects*. In addition, we required that the use of interaction objects be supported by some model for defining interaction.

The requirements for separation and programmability of the interaction lead to the use of scripting languages, i.e., interpretative languages used outside of the application. The requirement for an interaction model lead to the use of finite state machines (FSM's), in which events move the interaction from state to state. Each of these design decisions will be described in more detail below.

2.1 Modelling Interaction with Finite State Machines (FSM's)

In order to support interaction several approaches can be used, such as grammars, event handlers or state-transition networks. They differ in flexibility and expressiveness (see [5] for a more detailed comparison):

1. Grammars are not suitable for expressing multi-threaded interaction (such as invoking a help function at arbitrary points in the interaction).
2. The mechanism of event handlers, as found in window systems, consists of sending events to event handlers which can perform arbitrary actions. Although this is a general and powerful mechanism, it imposes little in the way of structure.
3. State-transition networks are often used in user interface management systems. One example is the Generic Window Manager GWM [8], which uses FSM's and a LISP dialect to specify the behavior of a window manager. See [5] for other examples. The main disadvantage of state-transition networks is the danger of state explosion, in which similar state-transition behaviors are to be repeated in several states. This disadvantage can be mitigated by the use of recursive state-transition networks. In addition, augmented transition networks extend these by adding computation facilities (data storage and functions, which are only visible within the network).

All in all, we have chosen the model of finite state machines, because it gives support for modelling interaction in terms of states and transitions. By using it in a programming environment, it becomes general and powerful enough to express even complex interactions.

2.2 Specifying Interaction using Scripting Languages

In order to be able to change the definition of an interaction dynamically, simply separating it from the application functionality is not sufficient. If interaction is defined separately, but changing it still requires reprogramming or recompiling part of the application, then much of the advantages of separation are lost. What is needed in addition to separation is an *interpretative* definition of the interaction. This approach is already used in other systems, such as Tcl [9].

Rather than defining our own scripting language, or relying on one specific scripting language, a general software interface for scripting languages was developed. Through a small set of functions the interaction object can execute files containing scripts, as well as call functions defined in a script. In practice, some requirements are imposed on a scripting language to be used in this way, but these requirements are easily met by most scripting languages (see Section 3.4 for details).

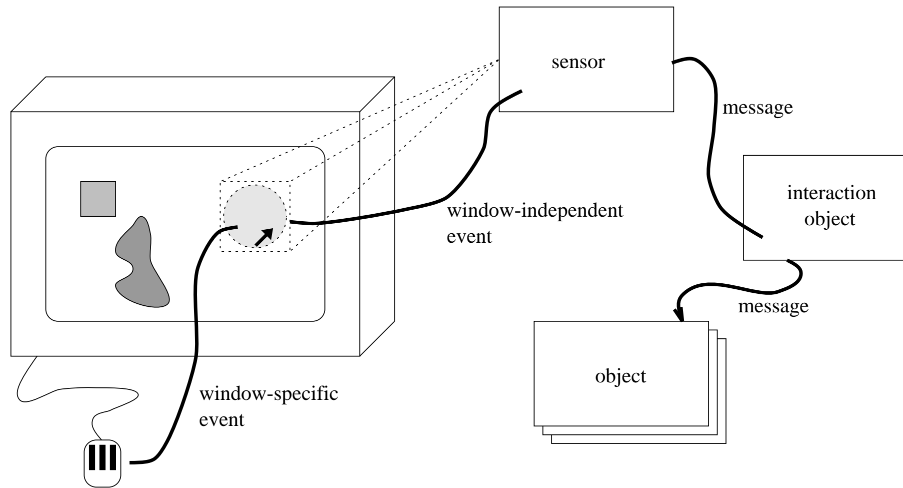

Figure 0.1: A sensor in an interactive application

2.3 Programmable Interaction Objects
Having this general mechanism of interaction objects as script based programmable FSM's opens up a way to reuse interaction objects. Interaction objects can be composed into more complex interaction sequences and can be reused as interaction patterns. Also, the mechanism of defining the behavior of an interaction object in a script, separate from the application, makes it possible to combine existing application functionality into new user-level functionality. Interaction objects can then be used to glue together suitable application building blocks into new applications.

The notion of programmable FSM's can also be used in contexts other than interaction. In the current draft of the new proposal for the ISO-standard PREMO (Presentation Environment for Multimedia Objects [4]), general controller objects are defined. The distinctive characteristic of controller objects is that they maintain a programmable state-transition model. In PREMO, controller objects are used for interaction as well as for synchronisation. For example, a controller object may model an OR-synchroniser, which takes some actions as soon as it receives one event out of a set of events, or an AND-synchroniser, which only takes some action after it has received all of a set of events.

One consequence of using programmable interaction objects is that the definition of interaction is separated from an application's functionality. Communication between these two sides is carried out through their respective software interfaces. For example, when using a mouse to drag an object displayed on the screen, the events coming from the mouse are not forwarded directly to the object involved, but rather to an associated interaction object. This interaction object then determines how to perform the dragging operation. It may, for example, call the 'move' method of the object on each event. Alternatively, it may perform the feedback itself by showing an outline of the object, and only call the 'move' method of the object when the drag operation is complete. The point is that direct manipulation and other user interaction now only use the functional interface of the objects involved.

Another point arising from the separation is that an association is set up that forwards events directed to objects to the appropriate interaction object. For this association, we use the notion of sensors (see also GoPATH [2]). A sensor is an object that may be associated to other objects. In addition, it is also associated to an interaction object. Sensors express to the window system an interest in receiving events. In fact, they are the only objects that do this. Whenever an input event is generated, the window system forwards it to the appropriate sensor (see figure 0.1).

A sensor expresses interest in a event by defining a sensitive 'region' and an event type. For example,

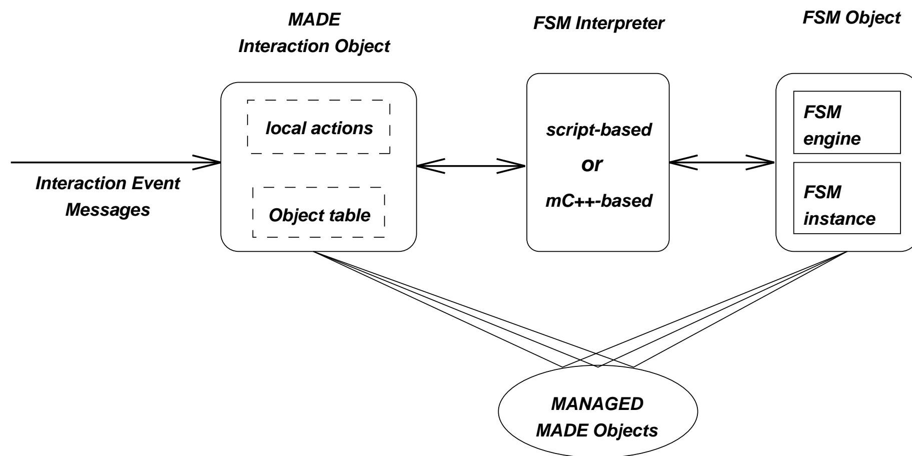

Figure 0.2: Interaction object architecture

if a sensor whishes to receive events directed to a graphical object, it can define its region to coincide with the area covered by the graphic object. Alternatively, a sensor could define a different region, for example, one which covers only the corners of a bounding box for the graphic object. The sensor sends the event to its associated interaction object by calling a specific method defined for the interaction object. The sensor is free to change the event, send another event, or send messages to other objects as well. Typically, the event is translated into a window system independent event, so that the objects receiving events do not depend on the specific window system that is used. This implies that a fixed set of events must be defined. Events are also (of course) objects, so any object could create event objects and send these to other objects, simulating input events.

The mechanism of sensors allows the establishment, dynamically, of a link between an object being manipulated and an interaction object. The object involved is unaware of its associated sensor(s).

3. Architecture

3.1 Overview

Assuming that an interaction object instance has been created, then the architectural structure depicted in Figure 0.2 will exist for it (details of the instantiation and initialisation of an interaction object are given later):

1. The interaction object itself, which receives messages representing interaction “events”, typically from an associated *sensor object*, but also potentially from other interaction objects in an interaction hierarchy.

2. The FSM object, which consists of a general FSM engine and a specialized FSM instance. The FSM engine provides methods for the creation of an FSM and for performing transitions on an FSM. The FSM instance defines local actions to be performed on recognition of a particular event in a specific state of the FSM.

   The entire FSM object is defined either in a scripting language or is `C++`-based. An FSM object is not required. If none is defined for a specific interaction object, any events sent to the interaction object are redirected using a set of virtual functions defined in the interaction object[1]. The default definitions of these virtual functions is to do nothing. They can be redefined by writing an appropriate subclass of the general interaction object class.

[1] In the `mC++` programming language used in the Made environment (see Appendix 1), these functions can also be dynamically redirected using *delegation*.

3. If the FSM object is defined in a scripting language, then also a script interpreter object is present. Conceptually, if the FSM object is `C++`-based, then this interpreter amounts to the `C++` runtime environment.

Both the interaction object and the related FSM object maintain equivalent tables of information on the objects that are managed by the interaction object.

The basic operation of an interaction object is as follows:

1. On receipt of an interaction event, the interaction object invokes its FSM object through an appropriate interpreter and passes it the event and any arguments. In case no FSM object is defined for the interaction object, it calls one of its own virtual functions, so that the response on events can be handled by a subclass of the interaction object.
2. The FSM engine determines the action to be performed for the event based upon the transition defined for that type of event in the current state. The engine may call a locally defined procedure within the FSM instance itself. Alternatively, it may either invoke a locally defined action method within the interaction object or invoke a public method of any other MADE object. These later two possibilities rely on the provision of the MADE dynamic call mechanism (see Section 3.3). Of course, a local action defined in an FSM instance may also make use of this dynamic call interface.

In MADE, the FSM that defines the behavior of an interaction object can be specified in three ways:

1. by defining a subclass of a general interaction class;
2. by providing a `C++`-based FSM;
3. by providing a script-based FSM.

The first alternative is the most restrictive, in that interactive behavior is compiled into the program and the programmer has to code all state manipulations. The second alternative adds support in the form of a general FSM class. The third alternative in addition allows interactive behavior to be defined outside of the compiled program, giving more flexibility in designing user interaction.

The specific approach taken, however, is completely hidden to the rest of the program that uses the interaction object. This means that, for example, one could first use a script-based approach to design some user interaction, and later incorporate a final version into the program so that it may run without the overhead of a script interpreter. All of this is without any change to the rest of the program.

3.2 Class Definition for the Interaction Objects

Figure 0.3 gives the `mC++` class interface for interaction objects (for a short overview on `mC++`, see Appendix 1). The functionality of an interaction object includes the following:

MInteractor(FSMname) :

The constructor's principal task is to create and initialise an FSM object, possibly with a script interpreter. The `FSMname` argument is used as the basis for locating the FSM to be used. It can refer to a file containing a script definition of the FSM. Alternatively, it may refer to a `C++` object implementing an FSM. This second possibility is checked using the dynamic call interface (see Section 3.3). If `FSMname` does not refer to an FSM definition, the interaction object is supposed to use its hard-wired behavior.

resetInteractor() :

Reset the state of the FSM associated with the interaction object. Because resetInteractor() is declared as virtual, subclasses can override this function to provide hard-wired behavior.

```
Mutex MInteractor {
public:
    MInteractor(const char * =0);
    virtual void resetInteractor();
    virtual void triggerInteraction(MEvent *, char * ="", MCva_list);

    void manageObject(MObject *, char *, MBool add);
    void delegateUnknownEvent(MObject *, const char *);
    void delegatePassEvent(MObject *, const char *);

protected:
    MSet managedObjs; // general set of ManagedObject instances
    virtual void unknownEventAction();
    virtual void passEventAction();

    ~MInteractor();
};
```

Figure 0.3: Interface of the general MInteractor class.

triggerInteraction(event, format, ...) :

Used by other objects to send an interaction event message to an interaction object. The format describes the types of additional parameters that may be present. This method is used primarily by sensor objects. Because this function is declared as virtual, subclasses can override this function to provide hard-wired behavior.

unknownEventAction() :

Method that is called whenever the interaction object receives an event that it can not handle.

passEventAction() :

Method that is called whenever the interaction object receives an event that it wants to pass on.

delegatePassEvent() :

delegateUnknownEvent() :

Methods that use the `mC++` delegation mechanism to delegate the two methods mentioned above, `unknownEventAction()` and `passEventAction()`, respectively, to another object.

manageObject(object, memberFunction, addOrRemove) :

Method that adds or removes a reference to another object, together with an identification of a member function of that object, to or from the set of objects managed by this interaction object.

3.3 The Dynamic Call Interface

The model of interaction objects has been implemented in the MADE project in a `C++` environment (see Appendix 1. The main addition made by MADE to `C++`, relevant to the implementation of the interaction objects, is the *dynamic call interface.*

The dynamic call interface allows methods of `C++` objects to be called, given only a string specification of the method to be called, and knowing only a common base type for all MADE objects. This mechanism is essential for allowing general FSM's to call arbitrary methods in arbitrary MADE objects. Also, it is needed in the implementation of the general scripting interface. The `C++` language does not offer this ability. Because of its static type checking, all object types and method signatures

must be known at compile time. C++'s mechanism of virtual functions allows runtime binding of methods to objects, but only for the set of functions that are defined, at compile time, to be virtual. These restrictions makes it cumbersome to implement applications that require runtime binding. For example, when accessing C++ objects from within a scripting language, the C++-method to be called will always be identified by the script using some interpretable description. In C++, this description then has to be matched with a method. In the MADE C++ environment (called mC++ [1]), we have added support for this. It is interesting to note that in the OMG specification [6] a similar facility is included. In MADE, the dynamic call interface contains the following:

MCallMethod(MObject *obj, const char *signature, ...) :

Call the method specified by `signature` on `obj`. A variable number of parameters, to be forwarded to the member function to be invoked, can be supplied. These are handled using the standard C `stdarg` functionality.

MCallMethodv(MObject *obj, const char *signature, va_list ap) :

This one is similar to the previous function, but takes a reference to a variable number of parameters, rather than the parameters itself. Again, this uses the standard C `stdarg` functionality.

MInqMethod(MObject *obj, const char *signature) :

Check whether `obj` defines a member function that has `signature` as its declaration.

3.4 Requirements on Scripting Languages

As mentioned before, the design of the interaction objects allows for three possibilities: hard coded without an FSM, hard coded but FSM-based, and an external interpreter with an FSM encoded in a script. In the current implementation, the main requirements on a scripting language to be used in this way, are that it is embeddable (i.e. a software interface is defined for it which allows access from within C or C++), and that it contains the notion of functions callable by a name. Candidates for scripting languages that are suitable for use as FSM interpreters include Perl [11] (the latest version is now embeddable), Python [10], and Tcl [9]. Section 4 gives a working example of a script-based FSM using Python. It defines a general FSM class which is specialized into a simple controller for a video player. The example also shows an equivalent FSM written in C++.

4. EXAMPLES

Two examples are given that illustrate the use of programmable interaction objects. The first example, a video player, is intended to show the merits of using an FSM to describe the states and transitions inherent to a particular interaction. The second examples is intended to highlight the use of sensors, and to show the ease with which the interactive behavior and end-user functionality can be changed by simply changing a script that is used for the interaction definition.

4.1 Example: A Video Player

The first example implements a video player with the usual play, stop, pause, forward and backward buttons. An FSM can be used to model the behavior of the player. In this example, several entities are assumed which, for brevity, we do not specify further. These are:

1. a user interface with buttons for play, stop, fast-forward, etc.;
2. a sensor associated with each button;
3. a video player object;

In addition to these, an interaction object and an FSM are defined. The interaction object's interface is given in Figure 0.4. It is defined as a Mutex object, which is an mC++ construction that

```
Mutex Player : public MInteractor {
protected:
    void unknownEventAction();
    void passEventAction();

    void issueStop();
    void issueBackw();
    void issueForw();
    void issuePlay();
    void issuePause();
    void releasePause();
public:
    Player(char * =0); // The parameter contains the FSM name
    // For hard-wired behavior, define the following:
    // void resetInteractor();
    // void triggerInteraction(MEvent *, char *, MCva_list);
};
```

Figure 0.4: Interface of an interaction object for a video player.

provides protection against multiple access by several threads at the same time (see Appendix 1). The interaction object defines methods for invoking the appropriate actions on the video player. For example, it defines a method `void issuePlay()` to start up a video player. Although in this example the methods do not have arguments, there is nothing that prevents them from doing so.

The FSM definition is given in Figure 0.5. Figure 0.5 actually shows two FSM definitions, one written in Python, the other in `mC++`. The interaction object given in Figure 0.4 can be used with either of these, without any further changes. As indicated by the comments in Figure 0.4, it can even be used without an FSM by redefining the interaction object virtual methods `resetInteractor()` and `triggerInteraction()`.

When one of the buttons in the user interface is pressed, the sensor associated with the button sends an appropriate message to the interaction object. For example, if the play button is pressed, the sensor sends a message containing an event with name `Play`. The interaction object forwards the event to the FSM. The FSM specification in Figure 0.5 specifies that when a `Play` event is received while the FSM is in state `STOP` (which we assume here for the example), then the member function `void issuePlay()` should be invoked on the interaction object[2]. To continue the example, if next the fast-forward button is pressed, the sensor associated with that button sends a `Forw` event to the interaction object, which forwards it to its FSM. The FSM is now in state `PLAY`, and on receiving the `Forw` event it calls the local function `stopAndForw`. This function first call the `issueStop()` method on the interaction object, followed by calling the `issueForw()` method on the interaction object. This shows how in a script existing application functionality (stop and forward) can be combined into new end-user functionality (stop-and-forward).

The FSM is now in state `FORW`. Suppose the FSM receives another `Forw` event. Because this event is not allowed in the current state of the FSM (only `Stop`, `Pause` or `Play` are allowed), the FSM calls the `unknownEventAction()` method on the interaction object which is supposed to handle any inappropriate events.

[2]In the Python FSM specification, member functions of the associated interaction object are denoted by strings, local actions within the FSM are denoted using the local function name. In the `mC++` FSM specification, methods defined in an interaction object or in the FSM are both denoted by strings. They are distinguished at runtime: first a runtime check is made to see whether the function given by the string is defined in the FSM, and failing that, whether it is defined in the associated interaction object. The ability to check at runtime the existence of member functions is supported by dynamic call interface defined in the MADE environment.

```python
# The module 'madeif' interfaces to MADE programs
import madeif
# The module 'MFsm' contains a general FSM class
import Mfsm

# class Video is a subclass of the general FSM class
class Video(Mfsm.MFSM) :
  # local actions, defined by class functions:
  def localInit(self) :
    self.pausedState = 'STOP'

  def stopAndBackw(self,arg) :
    madeif.callmethod(self.iao,'void issueStop()' , 'n')
    madeif.callmethod(self.iao,'void issueBackw()', 'n')

  def stopAndForw(self,arg) :
    madeif.callmethod(self.iao,'void issueStop()', 'n')
    madeif.callmethod(self.iao,'void issueForw()', 'n')

  def stopAndPlay(self,arg) :
    madeif.callmethod(self.iao,'void issueStop()', 'n')
    madeif.callmethod(self.iao,'void issuePlay()', 'n')

  def paused(self,arg) :
    madeif.callmethod(self.iao,'void issuePause()', 'n')
    self.pausedState=self.returnState()
    self.setState('PAUSE')

  def outOfPause(self,arg) :
    madeif.callmethod(self.iao,'void releasePause()','n')
    self.setState(self.pausedState)

  # The definition of the FSM itself:
  initState = 'STOP'
  transTable={
    'STOP' :{'Play' : ('void issuePlay()' ,'PLAY' ), \
             'Backw': ('void issueBackw()','BACKW'), \
             'Forw' : ('void issueForw()' ,'FORW' )},\
    'PAUSE':{'Stop' : ('void issueStop()' ,'STOP' ), \
             'Pause': (outOfPause         ,       )},\
    'PLAY' :{'Stop' : ('void issueStop()' ,'STOP' ), \
             'Pause': (paused             ,       ), \
             'Backw': (stopAndBackw       ,'BACKW'), \
             'Forw' : (stopAndForw        ,'FORW' )},\
    'FORW' :{'Stop' : ('void issueStop()' ,'STOP' ), \
             'Pause': (paused             ,       ), \
             'Play' : (stopAndPlay        ,'PLAY' )},\
    'BACKW':{'Stop' : ('void issueStop()' ,'STOP' ), \
             'Pause': (paused             ,       ), \
             'Play' : (stopAndPlay        ,'PLAY' )}}
```

```cpp
Mutex Video : public MFSM {
private: char *pausedState;
public:  Video(MInteractor *anIao);
 void stopAndBackw();
 void stopAndForw();
 void stopAndPlay();
 void paused();
 void outOfPause();
};

MFSMEntry FSMspec[] = { // defintion of a video FSM:
    {"STOP"}, {"Play" , "void issuePlay()"   ,"PLAY"  },
              {"Backw", "void issueBackw()"  ,"BACKW" },
              {"Forw" , "void issueForw()"   ,"FORW"  },
    {"PAUSE"},{"Stop" , "void issueStop()"   ,"STOP"  },
              {"Pause", "void outOfPause()"  ,        },
    {"PLAY"}, {"Stop" , "void issueStop()"   ,"STOP"  },
              {"Pause", "void paused()"      ,        },
              {"Backw", "void stopAndBackw()","BACKW" },
              {"Forw" , "void stopAndForw()" ,"FORW"  },
    {"FORW"}, {"Stop" , "void issueStop()"   ,"STOP"  },
              {"Pause", "void paused()"      ,        },
              {"Play" , "void stopAndPlay()" ,"PLAY"  },
    {"BACKW"},{"Stop" , "void issueStop()"   ,"STOP"  },
              {"Pause", "void paused()"      ,        },
              {"Play" , "void stopAndPlay()" ,"PLAY"  },
    0
};

// The name of the start state and a reference to the
// FSM description are passed to the general FSM class
Video::Video(MInteractor *anIao)
      : MFSM(anIao, "STOP", FSMspec) {
    pausedState = "STOP";
}

void Video::stopAndBackw() {
    MCallMethod(iao, "void issueStop()");
    MCallMethod(iao, "void issueBackw()");
}

void Video::stopAndForw() {
    MCallMethod(iao, "void issueStop()");
    MCallMethod(iao, "void issueForw()");
}

void Video::stopAndPlay() {
    MCallMethod(iao, "void issueStop()");
    MCallMethod(iao, "void issuePlay()");
}

void Video::paused() {
    MCallMethod(iao, "void issuePause()");
    pausedState = returnState();
    setState("PAUSE");
}

void Video::outOfPause() {
    MCallMethod(iao, "void releasePause()");
    setState(pausedState);
}
```

Figure 0.5: Video player FSM written in Python (left) and `mC++` (right)

```
# The module 'madeif' interfaces to MADE programs
import madeif
# The module 'MFsm' contains a general base class for FSM's
import Mfsm

# class Dragger is a subclass of the general FSM class
class Dragger(Mfsm.MFSM) :
    mass = 0 # instance variable for storing the mass
    # local actions, defined by class functions:
    def makeHeavy(self, arg) :
        # arg[0] contains the line involved.
        mass = madeif.callmethod(arg[0], 'int getMass()', 'i')
        madeif.callmethod(arg[0],'void setMass(int)', 'n', [10000])
        madeif.callmethod(arg[0],'void makeInanim()', 'n')

    def setPos(self, arg) :
        x = arg[1]
        y = arg[2]
        madeif.callmethod(arg[0],'void setPos(int,int)', 'n', [x, y])

    def restoreMass(self, arg) :
        # arg[0] contains the line involved.
        madeif.callmethod(arg[0],'void setMass(int)', 'n', [mass])
        madeif.callmethod(arg[0],'void makeAnim()', 'n')

    # The definition of the FSM itself:
    initState = 'DEFAULT'
    transTable={'DEFAULT' :{'MouseDown': (makeHeavy  , 'DRAGGING')},\
                'DRAGGING':{'MouseMove': (setPos     , 'DRAGGING') ,\
                            'MouseUp'  : (restoreMass, 'DEFAULT ')}}
```

Figure 0.6: FSM for dragging animated lines.

4.2 Example: Interaction with Object-Based Animation

Our second example deals with an animation of bouncing lines. In this animation, each line is an autonomous moving object that collides with other lines as well as with the walls that enclose its space. Each line has an equal mass, so that when two lines collide they bounce off equally in their opposite directions. Furthermore, the user may select a line with the mouse and subsequently drag the line through the space. The selected line is bound to the mouse pointer (i.e. it no longer determines its own movement), so that when it collides with other lines the selected line itself remains in its position whereas the other lines bounce away. Of course, the selected line can not be dragged outside of the walls.

This example is implemented by an `mC++` class modelling an animated line. This `mC++` class is an *Active* class, meaning that is has its own thread of control which calculates the next position for the line and redraws the line. The space containing all lines is implemented as an `mC++` *Mutex* class, which means that it is automatically protected against concurrent access by several threads. For this paper, however, the implementation of the interaction is more important. Each line has a sensor associated with it, that listens for mouse events. Each sensor propagates a mouse event to the same interaction object. The interaction object stores which line is currently being manipulated. When the interaction object receives a mouse down event, it sends a message to the line involved that sets its mass to a very high quantity. The FSM associated to the interaction object then makes a transition to a state

that takes care of dragging the line. After a mouse up event the FSM returns to its initial state.

Figure 0.6 shows the FSM for this example, again written in Python. Note that because of the script-based definition of the FSM, it is very easy to change the interaction so that the selected line will get a different mass, changes its color, etc.

5. Conclusions

We have described our notion of interaction objects whose behavior is defined by a programmable finite state machine. The state machine can be either hard-coded (in `mC++`) or it can be programmed by an interpretive scripting language. This choice is transparent to the interaction object and the rest of the application.

Interaction objects can dynamically and transparently be bound to application objects through the use of sensors. In this way, the objects are not explicitly aware of user interactions. Sensors forward window system events to interaction objects, which use the functional interface of the application objects to manipulate them.

The combination of interaction objects that are both programmable outside of the application and that can be bound dynamically to application objects makes it possible to separate the definition of user interaction from the application, and allows for rapid prototyping. This is a necessary condition for being able to design user interfaces for interactive applications in an incremental way, which is regarded as a prerequisite for designing good user interfaces. This holds even more strongly for multimedia application, where the technical complexity of the application may overwhelm the proper design of the user interface.

1. Short Overview of `mC++`

`mC++` [1] is an extension of `C++` that has been developed as part of the Made project. Features added to `C++` include the following:

Active objects These are defined by a regular `C++` class definition, except that the `C++` keyword `class` is replaced by the `mC++` keyword `Active`. The active class definition always contains a `main()` member function that is executed for each newly created instance. This construct allows for active objects that have their own thread of control. Communication between active objects can be achieved using synchronous or asynchronous messages, the type of message being defined by the receptor. Methods of active objects can specify explicit accept statements for receiving messages. An active object handles at most one message at a time.

Mutex objects These are defined by a regular `C++` class definition, except that the `C++` keyword `class` is replaced by the `mC++` keyword `Mutex`. Instances of mutex classes are automatically protected against concurrent access by multiple threads.

Prototypes and Delegation The object-oriented model employed by languages such as `C++` and SmallTalk is that of classes and instances, in which a class may inherit from one or more other classes. An alternative model, used in languages such as Self, is that of prototypes and extensions. In this model, the concept of a class does not exist. Instead, to create new objects, a copy is made of another *prototype object*, and the copy is then extended with additional data and/or member functions. The new object then handles the extended part of its functionality by itself, and *delegates* the rest to its associated prototype.

In `mC++`, in addition to the `C++` class model, we have defined a more restricted version of the prototype-extension model. In `mC++` a class may define a prototype, and exactly one instance of this is automatically created at runtime. Regular instances of the class may define functionality in addition to what is defined in the prototype, and use delegation whenever functionality is used that is only defined in the prototype. Furthermore, any object may delegate one or more of its member functions to other objects. When a member function of an object is delegated to

another object, the delegating object is no longer aware of any invocation of its own version of the member function.

REFERENCES

1. F. Arbab, J. Davy, F.C. Heeman, I. Herman, O. Jojic, G.J. Reynolds, and M.M. de Ruiter. Specification of the MADE object model and of the mC++ language, December 1993. (version 2.0).
2. J. Davy. Go: A graphical and interactive C++ toolkit for application data presentation and editing. In *Proceedings of the 5th Annual X Technical Conference*, USA, January 1991. (also in The X Resource, 1, Winter 1992).
3. Esprit project proposal – MADE I – 6307 – Technical Annex, March 1992. (version 1.1).
4. International Organization for Standardization. Presentation environment for multimedia objects (PREMO). ISO/IEC JTC 1/SC 24/WG 6, OME 61, 1993.
5. M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.
6. Object Management Group. The common object request broker: Architecture and specification, 1991. OMG document number 91.12.1, Revision 1.1.
7. I. Herman, G.J. Reynolds, and J. Davy. MADE: A multimedia application development environment. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems (ICMCS'94)*, Boston, U.S.A, May 1994. IEEE.
8. C. Nahaboo. The X11 generic window manager – GWM manual, 1991. (version 1.7l), Koala Project, Bull Research.
9. J.K. Ousterhout. Tcl and the Tk toolkit, August 1993. Computer Science Division, University of California, Berkely.
10. G. van Rossum. Python reference manual, February 1993. CWI, Amsterdam.
11. L. Wall and R.L. Schwartz, editors. *Programming Perl*. O'Reilly, Sebastopol, 1992. (ISBN 0-93717564-1).