Case Study in the formal specification of the Simple Player graphical interface for playing QuickTime movies, using the ADC interactor model

Markopoulos, Panos

712

Case Study in the formal specification of the Simple Player™ graphical interface for playing QuickTime™ movies, using the ADC interactor model

Panos Markopoulos
Dpt. of Computer Science
Queen Mary and Westfield College
Mile End Road
London E1 4NS
email markop@dcs.qmw.ac.uk

Case Study in the formal specification of the Simple Player™ graphical interface for playing QuickTime™ movies, using the ADC interactor model

1. Introduction

A case study in the formal specification of graphical user interfaces is reported. It is a reverse engineering case study of the use of the ADC formal architectural model of user interfaces introduced elsewhere [1]. The ADC (Abstraction Display Controller) model is based upon the interactor concept.

An interactor is defined as an abstraction unit that manages communication between interface software components, in two directions: from user to application and conversely. An interactor may communicate with other interactors, if not directly with the user or the application. Generally an interactor is capable of:

1. interpreting input from the display side according to its display status
2. sending its result to the abstraction side
3. processing input it receives from the abstraction side and
4. outputting its display status towards the display side.

Apart from the display status an interactor maintains some local state - the abstraction. Interactors are a specialisation of the object oriented notion of an object that distinguishes between the directions of communication for the data they process and by distinguishing between the two components of their local state.

The aim of this exercise is to assess the ADC model in practice; it is not concerned with methodological issues of the design of user interfaces or with formulating expressions of usability. The latter issues are the subject of current investigation.

The reverse engineering study consists a first test of the modelling scheme; it tests for the expressiveness of the modelling scheme and provides some feeling for the ease of writing the specifications using the model. It involves of the formal specification of an existing user interface in accordance with the modelling scheme. The formal specification, is required to capture interactive behaviours at a level of abstraction much lower than one would intend normally for a formal specification. In this way, the model is tested thoroughly by ensuring that the specification is a faithful representation of the interactive behaviour that does not abstract away from aspects of the behaviour that are difficult to express. In the process of reaching the formal specification, the practicality of the modelling scheme is tested, and improvements to the model are suggested that make it more usable. The choice of the appropriate abstraction level is discussed further in the section concerned with the scope of the specification exercise.

QuickTime™ is a system-software extension for the Apple Macintosh. It allows application programs to work with media such as sound, video and high quality compressed images. Simple Player™ is an application program that uses QuickTime™ to play and edit movies. Simple Player™ is interesting for it provides a good mix of diverse interactions rather than a repetition of similar and very simple behaviours. Simple Player™ was selected in fulfilment of the requirements listed below that had been set a priori, for choosing the application-subject of the case study.

- The application should be easily available in the lab. In the course of the study this proved to be of outmost importance. Intricate contingencies between the various interactive dialogues are difficult to foresee and have to be re-examined intermittently throughout the specification activity.

- The application should have interesting behaviour in terms of the process algebraic framework that is tested i.e. the temporal aspects of the dialogue are most interesting. As a counter example, a form filling interface where fields are filled in any order does not have an interesting temporal element. In contrast, modification of the frame-rate of a movie while it is playing, and the modes in which this activity is enabled is a more appropriate test for the model.

- Interesting layout and graphical properties are not an important consideration in choosing the application. Their description is not a strong point of the ADC model. For example, form filling interfaces or graphical editors would be interesting test cases for other types of models more adept in capturing this type of properties.

- The example application should have a realistic size: it is not sufficient to study a simple interaction technique, or just one aspect of the behaviour of the system. The size of the case study will be an indication that the ADC model is capable of scaling up.

2. A brief description of QuickTime™

QuickTime™ is an Apple Macintosh System-Software extension whose functionality consists in recording, editing and playing back time based data. The presentation of the data may include audio and video modalities. Simple Player™ provides access to a subset of the QuickTime™ playback and editing facilities. This section summarises some concepts pertaining to the architecture of QuickTime™, the data it handles and its functionality, as described in [2]. Only those aspects of QuickTime™ relevant to the case study are covered in this description (the scope of the case study is discussed in a later section).

The architecture of QuickTime™ is shown in figure 1. Each of its components provides a set of functions for the management of a general class of QuickTime™ features. These can be used without a deep knowledge of the particular technology they support. For example some of the components are the image compression and decompression component, the movie controller component, the clock components etc.

Basic Concepts

QuickTime™ uses the metaphor of a movie to describe time-based data. The movie is a multiple layer hierarchical organisation of data, although the application that uses it need not be aware of this organisation. Simple Player, or other playback applications, access the movie data through a set of movie playback functions that belong to the Movie Toolbox functional interface.

An important element of movie data, is the specification of the time dimension: at what rate will the movie be played, for how long etc. A movie's time coordinate system provides the context for evaluating the passage of time in the movie. The time coordinate system defines an axis for measuring time, marked with a scale which defines the basic unit of measurement, the time scale. A movie time scale defines the number of time units that pass each second in a given time coordinate system. Each time unit is equal to (1/time scale) seconds.

A time coordinate system specifies a duration, which is the length of a movie data in terms of number of time units. A point in a movie is identified by the number of time units elapsed from the beginning of the movie. A movie is also characterised by the rate at which time passes for the movie. This specifies the speed and the direction in which time travels in a movie. Negative rate values will move backward through a movie's data and positive values will move forward.

Special clock components generate time information for the use of the Movie Toolbox. Clock components derive their timing information from some external source e.g. the Macintosh tick count, or some special hardware installed in the Macintosh computer to provide its basic timing. Figure 1, shows the relationships between an application, the movie controller component, the Movie Toolbox, and a clock component.

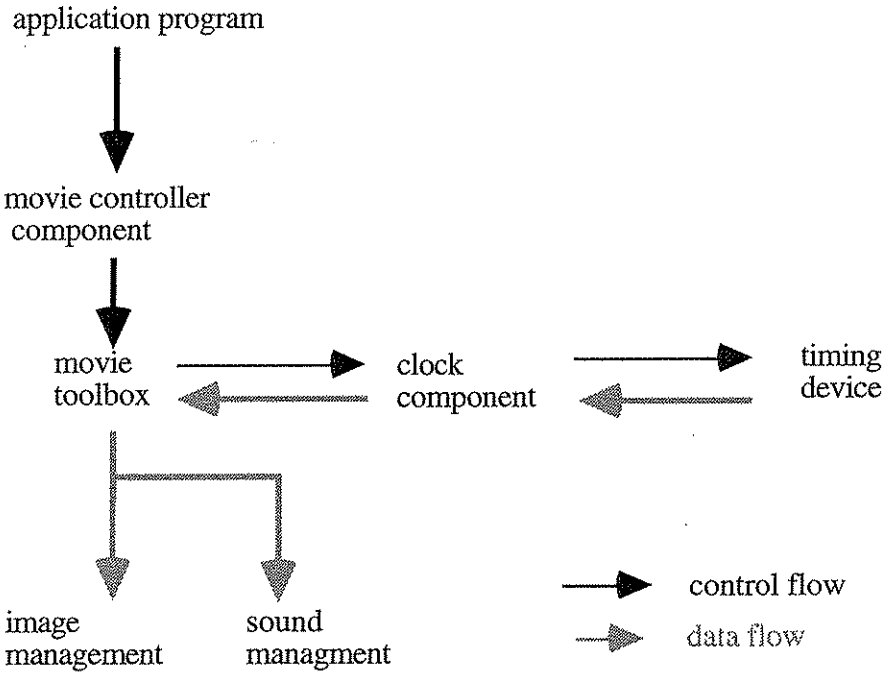

Figure 1. Relationships of an application, the movie controller component, the Movie Toolbox, and a clock component

Movie Characteristics

Each movie has its own time coordinate system and time scale. Other information related to the movie is: the movie's preview, the current position in a movie, the current selection, the active movie segment, the movie's display characteristics, preferred playback volume, preferred rate, current volume and current rate.

The current position in a movie is defined by the movie's current time. If the movie is currently playing, this time value is changing. When loading a movie from a movie file, the Movie Toolbox sets the movie's current time to the value found in the movie file. This value is updated each time the movie is saved. The current selection is specified by a start time (called selection time), and a duration.

The active movie segment is the part of the movie that the application is interested in playing. By default, it is set to be the entire movie. It may be changed to some segment of the movie—for example, in order to play a user's selection repeatedly. Setting the active movie segment, will have the effect that the Movie Toolbox uses no samples from outside of that range while playing the movie.

Display characteristics are defined by a group of display regions, and a set of functions to operate on them. The Movie Toolbox hides the intricacies of handling the display characteristics of a movie, via the functions *GetMovieBox* and *SetMovieBox* which are used to display a movie at a particular location on the screen. Finally a movie is associated with current and preferred settings for rate and volume. When a movie is started, its current settigns are set to their preferred value.

Movie Controller Components

The Movie Controller Component (figure 1), provides a functional interface which can support standard movie controller components provided by QuickTime™ for regulating sound, starting, stopping, pausing, single-stepping forward and backward, moving to a specified time etc. These components are capable of handling interaction with the user, but also package playback and editing facilities. Interactive applications may be written to include function calls to the Movie Controller Component functional interface. In fact, a shorthand for its functions is provided as an indexed set of actions. Programs may directly handle these actions, or let movie controller components handle them in a standard way.

Events handled by the movie controller components are associated with invocations of these actions or functions that actually return some value. A subset of the action set and the temporal orderings between them comprise the model of the functional core that is specified in later sections. Those actions only that are of interest within the scope of the reverse engineering exercise (as defined in paragraph 3) are discussed below.

- *mcActionPlay*. Play or stop playing a movie. Parameter data indicates the rate of play. Values greater than 0 play the movie forward; values less than 0 play the movie backward. A value of 0 stops the movie.

- *mcActionGotoTime*. Move to a specific time in a movie specified by a parameter.

- *mcActionGetMovieTime*. Returns the current position in the movie.

- *mcActionSetVolume*. Sets a movie's volume according to the parameter. This indicates the relative volume of the movie, ranging from –1.0 to 1.0.

- *mcActionGetVolume*. Return the current volume of the movie'.

- *mcActionSetSelectionBegin*. Set the start time of a movie's current selection. The parameter data specifies the starting time of the movie's current selection.

- *mcActionSetSelectionDuration*. Set the duration of a movie's current selection. The parameter data specifies the ending time of the movie's current selection.

- *mcActionSetPlaySelection*. Constrain playing to the current selection if the parameter is true, and cancel this option if the parameter is false.

- *mcActionGetPlaySelection*. Determine whether a movie has been constrained to playing within its selection.

- *mcActionSetPlayEveryFrame*. Instruct the movie controller to play every frame in a movie.

- *mcActionGetPlayEveryFrame*. Return a Boolean value indicating whether the movie controller has been instructed to play every frame in a movie.

3. Informal description of the interaction with Simple Player

Simple Player™ supports a mixture of interaction techniques:

- with the standard to Macintosh applications menu bar
- issue of commands by keyboard shortcuts
- graphical interaction techniques.

These interactions are briefly described below.

The menu bar groups three menus: The *File* menu provides access to the file system, the *Edit* menu provides standard editing operations on the movie data e.g. cut, copy, paste etc. The user may use the *Movie* menu to set parameters that will affect the movie play back, e.g. loop/loop backwards and forwards, play selection only, set and go to the 'movie frame' and finally setting the size of the display to one of four standard sizes. Alternatively the user may get information about the movie using the get-info from this menu.

Keyboard shortcuts may be used to invoke play and pause commands, or even as shortcuts to the menu interaction. Alternatively, the keyboard is used as a modifier of mouse input.

Since the ADC model is primarily focused on modelling graphical interaction, the above menu based interaction and strictly keyboard commands will not be of concern in this case study. The graphical interaction, which will be the subject of the specification, is described more thoroughly below.

Graphical interaction with the movie controller

The movie controller is a complex interactor offering the functions listed below. Along with each function the interactions that invoke it are described.

- Setting the volume. The volume control allows the user to adjust the sound volume. A volume slider is displayed when holding down the mouse button with the cursor on the volume control. The user may change the sound volume while the movie is playing. Also, sound can be muted by option-click on the volume button.

- Starting the movie. The play/pause button allows the user to start and stop the movie. Clicking the play button causes the movie to start playing; in addition, the movie controller component changes the play button into a pause button. Clicking the pause button causes the movie to stop playing. If the user starts the movie and does not stop it, the movie controller plays the movie once and then stops the movie. Playing may be achieved also by double-clicking on the movie image or indirectly by controlling the movie speed.

- Stopping the movie. The movie will pause by clicking on the movie image, or the play-pause button or indirectly by setting the movie speed to zero.

- Play backwards. Pressing the 'Command' key and pushing the 'Step Backward' button. It may also be achieved indirectly by controlling the movie speed.

- Control of the movie speed. By control dragging on the forward and backward step arrows. Option-play will make the application play every single frame.

- Displaying a particular frame, i.e. choosing a particular moment in the movie may be done randomly with the play bar, or by stepping ahead or backwards with the right and left pointing arrows. The play bar and its indicator (the thumb) provide real time access to any moment in the movie. The user may 'jump' to the beginning or the end of the movie with option-step forward or backward. This will also interrupt playing. With the step buttons the user can move through the movie frame by frame, either forward or backward. Holding the mouse button down while the cursor is on a step button causes the movie controller to step through the movie, frame by frame, in the appropriate direction. The play bar (a slider) allows the user to quickly navigate through a movie's contents. Dragging the thumb within the play bar displays a single frame of the movie that corresponds to the position of the indicator. Clicking within the slider causes the indicator to jump to the location of the mouse click and causes the movie controller component to display the corresponding movie frame.

- Defining a selection. If the shift key is pressed on the keyboard, the current position in the movie will be used to define a selection. The selection is shown as a black band on the play-bar. The endpoints of this selection may be changed by dragging the thumb, playing the movie, or stepping forwards and backwards through the movie. The user may unselect by clicking in the play bar away from the thumb.

- Changing the window size by dragging the size box. Keyboard modifiers will allow the window to maintain its shape (the shift key) or step through optimal shapes in four different sizes (the option key).

- Typical window behaviours like dragging and closing the window are allowed. Further user input may control the selection of the active window.

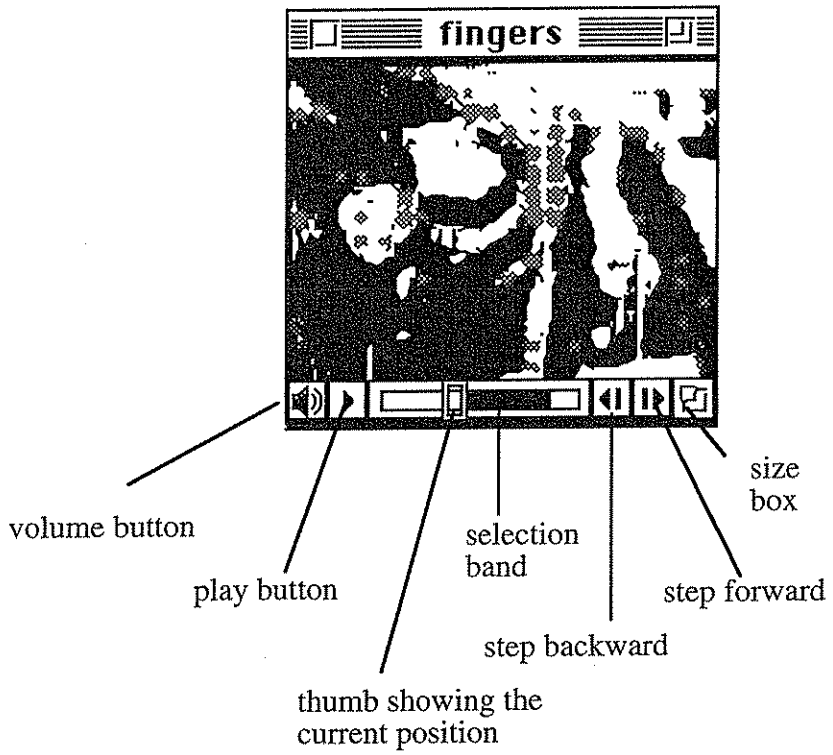

Figure 2. An instance of the Simple Player™ function. Some of the movie controller components are indicated.

Simple Player™ provides just a small subset of the functions of QuickTime™. The interesting characteristic of the interface is that this group of functions is invoked in a multitude of ways. Further these are modal, in the sense that the effect of user input action will vary depending on the (dialogue) state of other interactions . Thus the temporal ordering of interactions is challenging to model. QuickTime™ functions accessed through the Simple Player™ graphical interaction are described below in terms of inputs and outputs to the underlying application.

Input.

- Setting volume level.
- Start/pause play.
- Go to a frame.
- Set selection .
- Play all frames command.
- Set movie rate.

Output.

- Provide movie info to the interface; e.g. the current time indication, the duration of the movie, the current volume, or selection etc.
- Show a static frame.
- Play movie in whole or part. The rate of image is synchronised with a clock according to the current setting of movie speed, and the volume of the sound is set according to current setting.

4. Scope of the specification exercise

The scope of the application of the ADC model is the interface software only. This scope is depicted schematically in figure 3, mediating between a workstation agent, and the QuickTime™ system extension. The left boundary, with the QuickTime™ functional core, is defined by the set of the movie controller actions, which is a shorthand for accessing the Movie Toolbox function interface.

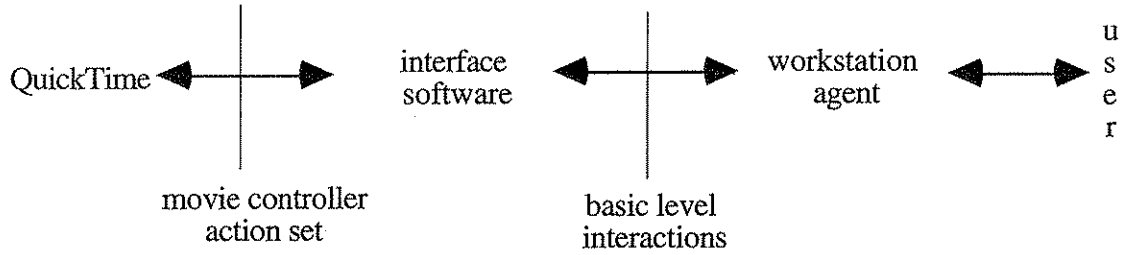

Figure 3. Scope and boundaries of the specification of the interface software.

The user has access to the interface through the physical interaction with the workstation agent. No assumptions are made about the basic level interactions other than their level of abstraction. Input events are considered as possible stimuli to the interface components and

no assumption is made about their source. For example mouse events over different interaction objects are considered independently as stimuli to the presentation object they refer to. The event management mechanism that would relate all these events to their source (the mouse in this case) is not specified. The same holds for output. Output events are not grouped in some way, and nothing is modelled about the destination of output data which could have been some model of the display.

The specification is concerned with an unusually low level of abstraction. In general a specification of the architecture of an interface will necessarily contain more implementation bias than an abstract model of the interface behaviour e.g. [6]. In the case of the reverse engineering example, even lower level of abstraction details are described so as to result in a concrete enough description that may convincingly be claimed to model faithfully the observable behaviour. Lowering the level of abstraction required for the reverse engineering example, increases the demands on the expressive power of the model, and as a result increases our confidence in its power. It is done though at the expense of the magnitude of the study. More sizeable exemplar software can be modelled if a higher level of abstraction is adopted.

Modelling at this low level of abstraction is not what would be required in a forward engineering study. However, by expressing such detailed behaviour, the ADC model is put to test and there is a concrete criterion for the success of the specification exercise. It can be claimed that the interaction is specified precisely, and difficult issues are not brushed over.

5. The specification process

A thorough study of the interaction with Simple Player™ was carried out, assisted by the on-line mini-manual. An initial description of the interaction with Simple Player™ was put together and used throughout the case study. Inconsistencies and errors in this description were corrected during the specification when thorough experimentation through simulation, with the application and its behavioural specifications was carried out.

The information regarding the software architecture of QuickTime™ and its related components, was used to define the scope of the exercise, and to draw the line that separates the interface from the application. This separation may not exist. In fact, contrary to the original expectations, it was observed during this specification that the boundary cannot be uniformly described as the action set of movie controller components. Some movie toolbox functions are accessed directly, as is the case with *gotoStart* and *gotoEnd* functions for positioning the player at the beginning or end of a movie. This function is not feasible to implement through the action set, unless trivially simulated with stepwise movements. Also setting the rectangle region in which the movie is displayed is accomplished with a movie controller function.

A list of movie controller actions and functions compiled provided a protocol in terms of input and output to the application. The temporal behaviour was observed by experimentation with Simple Player™. Examples of such temporal behaviour are: changing the volume while the movie is playing, the fact that after the movie has been stopped a still frame is displayed corresponding to the current position etc.

A formal description of the functional core was produced. Two versions of it were studied: one modelling the passage of time units and the other modelling only the resulting non determinism. The initial specifications of the functionl core were improved and corrected upon after their initial construction, as a result of testing with the interface specification.

A design of the specification was constructed in terms of communicating interactors, the data and signal flow between them. This design consisted an initial hypothesis as to how

the interaction software could be structured in terms of ADC interactors, to support the observed behaviour. This observation is determined by the level of abstraction adopted, and the nature of the phenomena modelled (temporal ordering of the input and output events). The defined structure refers only to the specification: it is not at any moment supposed that the actual software is structured in this way. The specification that results from this exercise demonstrates the feasibility of modelling complex behaviours in terms of instances of the interactor model connected appropriately.

Specifications of interaction components were individually tested and integrated into the specification one by one. During this process the initial structure of the specification was revised, with the addition of interactors that had not been foreseen initially, the decomposition of complex interactors to a group of simpler ones, and with unexpected dependencies between interactors. Further, the exercise revealed elements of behaviour that are hard to capture in the model: extensions and special cases were developed and adopted in the course of the specification, that are consistent with the ADC model, and do not compromise abstractness or modularity. These are discussed in the closing sections of this report.

When interactor specifications are connected to the functional core, its temporal behaviour is constrained. For example, the output of a frame is only possible if the display interactor is able to receive such data. Slowly all gates of the functional core were constrained by interactors, and interactors were added in a 'top down' fashion, proceeding from those interacting directly with the application and finally specifying those directly controlled by the user.

Each interactor was specified after its connectivity with other components of the specification was fully worked out. The data handled by the interaction was specified trying to answer some simple questions:

1. does it have a display?

2. does it provide input to the application or to other interactors?

3. what is the sort of the data connections?

4. is the interpretation of input received by the interactor dependent on its display?

5. what temporal ordering can the interactor induce on the events on its gates?

Data types, were specified along with their corresponding interactors. Provided an interactor has some status components, either a display or an abstraction element, it will be associated with a data type By deciding upon the sorts of the display and abstraction data, and given that connections are typed, the signature of the interactor data type follows by substitution from the general definition of the interactor data type described fully in [1].

If the interactor has an abstraction status, it should at least have an input and a result operation. If it also has a display status, then the input operation uses the display status in interpreting the input. Further, it will have an echo operation to update the display on the receipt of input, and a render operation to translate input from the abstraction side. In short, the signature of the interactor may be derived mechanically by defining the sorts of the status components of the interactor, and the sorts of the data that is input and output to it. Semantics may be given to the specification by adding equations to the data type. Special cases were developed for the cases where one of these components was nil.

When input comes from other interactor components it is not always clear to the specifier, whether it should be considered as coming from the abstraction side or the display side. For example, all interactors with a display component are modelled as having an input gate *iaGc*, from which position information is received. In the specification it is considered that

this information comes from the resize-box interactor; it does not concern us if this is in fact the case in the actual implementation. It is not clear if the resize box is at the abstraction side (higher level of abstraction) or the display side (lower level of abstraction). The real question of interest to the specifier and one that is more easily resoved, is whether or not the input value received by the interactor is interpreted on the basis of the current display status. This is the crucial difference between input received on gate G_{ipp} and an input received on a gate G_{ia}. In the example of the resize box, the result of updating the display to accommodate the new position of the resize box, is independent of the current abstraction of the movie player. Thus interpretation of input is handled better by the two separate interpretation functions *receive* and *render*, and as far as the modelled interactors are concenred the resize box is received at the application side.

The specifications of the data types managed by an interactor are not complete. A few equations were written to make the whole specification more meaningful during the simulation. This was also done in a regular fashion. The interactor's own data type is written as an extension to data types describing some of the required data operations on display or abstraction alone. For example, the interactor that sets the volume, uses its own data type *volume_ad*. This is an extension of a *popUpSlider*, which defines a graphical entity that maps points to integers, and vice versa and records the current display area (a rectangle), and the current position of the slider-thumb. Interactor display and abstraction status specifications are built from general-use components as would be always the case in algebraic specification.

Temporal orderings to the behaviour were described in the *run* sub-process of the controller component. Where possible a constraint oriented style was adopted. In other cases (e.g. a controller handling lexical level interaction), a simple state oriented specification of the dialogue was preferred for clarity. In the *run* process, all LOTOS language constructs are possible to use, although as a matter of style enabling and disabling was avoided in most cases. Still, all specifications produced in this exercise can be written in terms of prefix and parallel composition.

In part, defining the dialogue constraints imposed by a particular interactor, can appear arbitrary. Consider dragging an indicator-thumb over the player bar. Dragging involves pressing, moving and releasing the mouse. These interactions individually modelled because the evoke different feedback from the interface, and at times different application functionality e.g. dragging the mouse inside and out the play/pause button. It is also obvious that the mouse cannot be pressed twice without being released. This constraint is modelled as an input constraint on the relevant interactor, since there is no model of the physical input devices. Such constraints could be superfluous if a model of the workstation agent is constructed prior to the specification. Indeed if the ADC model is used as a design tool, it will be most economical to define such a lower-bound agent too.

A diagrammatic representation of the structure of the specification was used, and improved upon throughout the exercise. It is used with minimal explanations for the presentation of the specification in this report; it is intended that it is formally defined as a visual representation of the ADC model. Finally standard sub-components were revealed, and it is interesting to examine how they can be identified to be used as a library of specification components.

The remaining of this report presents the product of the specification exercise. The model of the functional core is presented first followed by the interactors: their behavioural specification and the specification of their data type. In the final section, some of the points raised above, that are themselves the expected result of a reverse engineering case study, are discussed more closely.

6. The functional core specification

A specification of the externally observable behaviour of the application functionality accessed through Simple Player™ (for short it is referred to below as the *functional core*) consists a bootstrapping activity for the specification of the interaction. The formal description of this functionality, in a form compatible with the modelling approach used for the interface software defines unambiguously one of the two bounds of the scope of the interface software (as defined in section 3). First an abstraction of the movie data is modelled.

Modelling Movie Data

A movie is associated with some state information. The components of this state information that are of interest within the scope of the specification exercise are:

- the current position in a movie, defined by the movie's current time and the movie duration.
- the current selection, defined by a start time and its duration.
- the active movie segment, defined by a start time and a duration. The active movie segment is the part of the movie that the application is interested in playing. By default, it is set to be the entire movie; it may though be set to some segment of the movie.
- the controller boundary rectangle and the movie box, both defined by a rectangle. For this study they are manipulated by moving the resize box. The movie *display box* defines a rectangle on the screen where the movie is displayed,. set with the movie toolbox function *setMovieBox.*
- the current playback volume.
- the current playback rate.

A movie is associated with preferred settings for playback rate and volume. These settings represent the most natural values for these movie characteristics. When the Movie Toolbox starts to play a movie it uses the preferred values for rate and volume.

Movie data is modelled as a LOTOS data type. Its signature is included below. The full specification can be found in the annex 1, where a full listing of the specification code is provided.

```
type movieData is segmentType, graphics
sorts movie, frame, still, sound, MData
opns
      movieTime          :     movie              ->    Int
      duration           :     movie              ->    Int
      volume             :     movie              ->    Int
      rate               :     movie              ->    Int
      selection          :     movie              ->    segment
      activeSegment      :     movie              ->    segment
      allFrames          :     movie              ->    Bool
      movieBox           :     movie              ->    rct
      setMovieTime       :     movie, Int         ->    movie
      setDuration        :     movie, Int         ->    movie
      setVolume          :     movie, Int         ->    movie
      setRate            :     movie, Int         ->    movie
      setSelection       :     movie, segment     ->    movie
      setActiveSegment   :     movie, segment     ->    movie
```

```
    setMovieBox      :    movie, rct      ->    movie
    incr             :    movie           ->    movie
    decr             :    movie           ->    movie
    getMoviePict     :    movie           ->    still
    videoFrame       :    movie           ->    frame
    setAllFrames     :    movie, Bool     ->    movie
eqns
(* equations ommitted below *)
```

endtype

A *movie* is a sort, that holds complex data, whose values are returned by inquiry operators. It can be thought of as a tuple that consists of:

- an integer value describing the current time of the movie, returned by the inquiry operation *movieTime*, and set by the operation *setMovieTime*.
- an integer value describing the duration of the movie, returned by *duration.*, and set by *setDuration*.
- an integer value describing the current volume, returned by *volume*. and set by *setVolume*.
- an integer value describing the current rate, returned by *rate*. and set by *setRate*.
- a segment (described as a pair of integers) describing the current selection, returned by *selection*. and set by *setSelection*.
- a segment describing the active segment, returned by *activeSegment*. and set by *setActiveSegment*.
- a Boolean flag returned by *allFrames* and set by *setAllFrames*.
- and the sort *movieData*, which represents the hierarchical movie structure. The inquiry operations *getMoviePict* and *videoFrame* may be used respectively to return a still frame or the video and audio information indexed by the current movie time. (the fact that sampling rates are different for audio and video signals is abstracted away: it is sufficient to model that both data is dependent on the current time).

Apart from the constructor operations used to set the values of the sorts of the data type, two more constructors are defined: *increment* and *decrement*.. They are used as a shorthand for moving the time forwards or backwards by one.

The equations of the data type, which can be found in the full listing of the specification, provide a complete specification of the data type: the equations specify the result of applying each inquiry operator to all the constructors. The result of applying each constructor is constrained so that it will only change the value returned by the corresponding inquiry operator.

The choices made for the description of the movie data are definitive for the level of abstraction for the specification of the interface. The description is consistent with the architecture as defined in [2], but also aims to be as simple as possible. For example, the data that is presented to the user when a movie is playing, may include both video and audio data. The two streams of data can be considered independently; in this case their synchronisation would have to be modelled explicitly. Alternatively, this distinction can be abstracted away from, by considering a single atomic stream of data that contains audio and video information. The choice impacts the way still frames are described. If the two streams are modelled, the still image may be one sample of data from the input stream. If the more abstract option is adopted, a still image is some data indexed by some time value.

The obvious connection between the continuous video signal and the still frame is not explicit. This approach was preferred as it makes the interface description simpler.

The behaviour of the functional core

The functional core may

- play a movie forwards or backwards. It may do so at a specified rate, possibly missimg some frames to sustain the required rate, or at the other extreme playing every frames and synchronising with the display, irrespectively of the rate value.
- show a still image from the movie given the position of the frame in the movie in time coordinates.
- allow the interface to set or access the state of the movie, any of the components of the 'tuple' that describes it: the position of the current frame, current volume, current rate, current active segment etc.

The specification provides only an abstract model of the functional core, given that the focus of this research is the specification of the user interface. The specification of the functional core follows the extended automata style [4]. The behaviour is described by transition between certain 'meta-states' of the player; the transitions are modelled by the instantiation of the corresponding behaviour expression.

- initial state
- with the video player idle, i.e. not playing
- playing forward
- playing backward.

These meta-states group many interactions. In the inital (meta-) state, the functional core can only be initialised, and associated with a movie. Transitions between these 'meta-states' occurs when the movie starts or stops playing and when the direction of play changes. Otherwise, the meta-states group manipulations of movie state information, described by the parameter M of sort *movieData*. The play-forward and play-backward processes also have a Boolean parameter, the flag *ready*. This flag captures the dependency on time for the behaviour of the application. The functional core will set *ready* to true when it is ready to output a new frame, it will it to false when it is unable to display new output.

The only architectural decision is that *fnctCore* has several gates:

- all gates prefixed by the word **'action'** which are used to issue commands to the application.
- **out_ok** used to synchronise with the component that displays the images
- **video** for sending back the sound and pictorial output (video)
- **data** for sending back information

Intuitively, modelling time seems to be an issue that will dominate the interaction with the movie interface. However real-time issues e.g. what is a click and what is a double-click in

the interface, can be considered to be implementation details. In the case of displaying video which induces tight constraints on the synchronisation with the screen, time is a source of non-determinism for the design level specification, and a model of real time is not necessary at this level of specification.

The reduction of timing issues to the non-determinism they introduce is demonstrated with the construction of two versions of the specification. The first models the synchronisation with a clock, and the latter abstracts away from this issue, and provides a simpler view of what the functional core does. The synchronisation with the clock is hidden from the user interface. Thus while an object definition of a clock provides a model of the passage of time, the two versions of the specification are observationally equivalent. In the second model, there is no clock entity, but only the non-determinism introduced by the consideration of time is represented.

Timed model specification

In the timed model the passage of time is represented by a clock, with which the *videoPlayer* process synchronises. The passage of a time unit is signalled by an event **tick** of the **Clock** process. Its meaning in terms of real time (e.g. seconds) is an implementation concern. As mentioned in the section describing the basic concepts of QuickTime™, the time unit is defined in terms of 'real' time, but this definition is only of interest to the clock components of QuickTime™ which synchronise with a clock device. For the other components of QuickTime™, and applications that play movies, it is enough to model the passage of time units.

The Clock process is a high level abstraction of the quite complex Clock component of QuickTime™, that simply models the passage of time as a sequence of events, corresponding to a passage of a unit of time for the movie. It is defined as follows:

```
process Clock[t](theTime: Int): noexit :=
      hide tick in
          ((t!theTime; Clock[t](theTime))
       []   tick; Clock[t](s(theTime)))
endproc
```

The adopted 'extended automata' style results in a state oriented specification. The state of the process is encoded partly in the parameter M (monolithic style) and partly by specifying which process is triggered by each event (state oriented style). An event may trigger any of three processes: *videoPlayer*, *playF* and *playB*. The process *playF* represents the 'state' of the functional core where it is playing the movie in the forward direction, i.e. with a positive integer value of the current rate. *playB* corresponds to a negative rate and *videoPlayer* to rate 0, i.e. an idel movie.

These processes synchronize with the clock via the gate t. From this gate they receive the number of ticks counted since the initalisation of the process *functionalCore*. Processes *playF* and *playB* are instantiated with a timing parameter. This corresponds the time slot in which the next video output event should occur. It is discussed extensively below, how the maniulation of this value by *playF* and *playB* specifies how the video output is output at the specified rate.

Process *videoPlayer*, but also *playF* and *playB* may receive an*actionPlay* event from the user interface. This is associated with a rate value. If this is 0, then if the movie is playing it will stop, and the system returns to the state described by process*videoPlayer*. If the rate is negative *playB* will be triggered; with a positive value*playF* is triggered. When playing is stopped, the functional core will output the frame at the current position before returning to the initial state (*videoPlayer*).

The other interactions with videoPlayer consist in setting flags or managing inquiries about state infomration held by the movie data in parameter *M*.

VideoPlayer is defined as follows:

```
process videoPlayer[...]
            (M:movie):noexit :=
      actionPlay?rt:Int [0 lt rt];
      t ?tm:Int;
      playF[...](true, setRate(M,rt), tm+rt)
[]
      actionPlay?rt:Int[rt lt 0];
      t ?tm:Int;
      playB[...](true, setAllFrames(setRate(M,rt),false),tm+rt)
[]
      actionPlay?rt:Int [rt eq 0] ;
      video !getMoviePict(M);
      videoPlayer[...](setRate(M,0))
[]
      actionPause;
      video !getMoviePict(M);
      videoPlayer[...](setRate(M,0))
[]
      actionGetMovieTime;
      data !movieTime(M);
      videoPlayer[...](M)
[]
      actionGotoTime ?targetPosition:Int;
      video !getMoviePict(setMovieTime(M,targetPosition));
      videoPlayer[...] (setMovieTime(M, targetPosition))
[]
      gotoBeginningOfMovie;
      video !getMoviePict(setMovieTime(M,0));
      videoPlayer[...](setMovieTime(M, 0))
[]
      gotoEndOfMovie;
      video !getMoviePict(setMovieTime(M,duration(M)));
      videoPlayer[...](setMovieTime(M, duration(M)))
[]
      actionSetSelection ?selection:segment;
      videoPlayer[...](setSelection(M, selection))
[]
      actionSetAllFrames;
      videoPlayer[...](setAllFrames(M, true))
[]
      actionSetVolume ?newVolumeLevel:Int;
      videoPlayer[...](setVolume(M,newVolumeLevel))
[]
      actionGetMovieVolume !volume(M);
      videoPlayer[...](M)
[]
      mcSetMovieBox?R:rct;
      videoPlayer[...](setMovieBox(M,R))
endproc
```

As mentioned above playF describes the movie when it is playing forward. Alternatively *playF* might answer a query about positioning information, and continue its normal behaviour. As with the listing of *videoPlayer* lists of gate identifiers are omitted from the process instantiations.

```
process playF[...] (ready:Bool, M:movie, W:int): noexit :=

      [ready and (movieTime(M) le duration(M))] ->
      video !videoFrame(M);
```

```
      data !movieTime(M);
      playF[...](false, incr(M),W)
[]
      [ready and (movieTime(M) eq duration(M))] ->
      video !videoFrame(M);
      out_ok;
      data !movieTime(M);
      videoPlayer[...](setAllFrames(M,false))
[]
      [ready]      -> t?tm:Int [W le tm];
      (hide missingFrame in
            ([not(allFrames(M))] ->missingFrame;
            playF[...](ready, incr(M), W+rate(M))
      []
            [allFrames(M)] ->missingFrame;
            playF[...](ready, M, W)))
[]
      [not(ready)] -> t?tm:Int [W le tm];
      playF[...](ready, M, W+rate(M))
[]
      actionGetMovieTime;
      data !movieTime(M);
      playF[...](ready, M, W)
[]
      actionGotoTime ?targetPosition:Int;
      video !getMoviePict(setMovieTime(M,targetPosition));
      videoPlayer[...]
            (setMovieTime(M, targetPosition))
[]
      gotoBeginningOfMovie;
      video !getMoviePict(setMovieTime(M,0));
      videoPlayer[...]
            (setMovieTime(M, 0))
[]
      gotoEndOfMovie;
      video !getMoviePict(setMovieTime(M,duration(M)));
      videoPlayer[...]
            (setMovieTime(M, duration(M)))
[]
      out_ok;
      playF[...](true, M, W)
[]
      actionSetVolume ?newVolumeLevel:Int;
      playF[...] (ready, setVolume(M,newVolumeLevel), W)
[]
      actionGetMovieVolume !volume(M);
      playF[...](ready, M, W)
[]
      actionPlay?rt:Int [0 lt rt];
      t ?tm:Int;
      playF[...](true, setRate(M,rt), rt+tm)
[]
      actionPlay?rt:Int [rt lt 0];
      t ?tm:Int;
            playB[...] (true, setRate(M,rt), tm+rt)
[]
      actionPlay?rt:Int [rt eq 0];
            video !getMoviePict(M);
            videoPlayer[...](setAllFrames(setRate(M,0),false))
[]
      actionPause;
            video !getMoviePict(M);
            videoPlayer[...](setAllFrames(setRate(M,0),false))
```

```
[]
      mcSetMovieBox?r:rct;
      playF[...] (ready, setMovieBox(M,r), W)
endproc
```

The process is instantiated with a parameter W specifying the time interval within which a video output event should be generated. Flag *ready* represents the state of the display resource. If it is set to false, that means that the display, and therefore the interface is not yet ready to receive a new frame.

If the display is ready then the movie may output the next frame. If this happens to be the last output event, when

```
movieTime(M) eq duration(M)
```

then the movie stops playing after it has played the last frame and synchronized with thedisplay, over gate *out_ok*. Otherwise, the next frame is output and *playF* is enabled recursively, but this time with the flag ready set to *false*. this can only be reset by a synchronisation with the interface over gate *out_ok*, after the output has been displayed.

A syncrhonisation with the clock may happen before the output occurs. It could be for example because decompressing a video frame is a slow process. This incurs a time delay, during which synchronisations with the clock occur. The time is read, and its value is compared with the time slot in which the frame should take place.

As long as there is still time before the time slot expires:

```
tm lt W
```

then nothing happens. playF waits for either the output event to happen (as described above) or for the time slot to elapse

```
W le tm
```

If this happens, then the frame has delayed. We distinguish two cases:

1. the application is set so as to play all frames, then a hidden event missing frame occurs, before the playF is instantiated again, with the same wait variable W.

2. the application will skip a frame in order to maintain the required rate. Then the movie position is incremented by one and the value of the wait parameter is increased by the value of rate of themovie[1]

Untimed model specification.

In the untimed version, the process *fnctCore* consists simply in the triggering of *videoPlayer* by the *init* event, and a clock component is not modelled. In the listing of process *videoPlayer* below the gate names are omitted in recursive calls for the sake of brevity. Processes *playF* and *playB* are called with the same gate set as *videoPlayer* itself.

```
process videoPlayer[...] (M:movie):noexit :=
      actionPlay?rt:Int [0 lt rt];
            playF[...] (true, setRate(M,rt))
[]    actionPlay?rt:Int[rt lt 0];
            playB[...] (true, setAllFrames(setRate(M,rt),false))
[]    actionPlay?rt:Int [rt eq 0] ;
```

---

1 This scheme model in a snchronous language a clearly temporal issue, that of ensuring that a movie is plauedwith a fixed ratee. In fact what is represented is that the occurrence of an event may be modelled before or after a synchronsiation with the clock (not together - truely concurrent events are not allowed inLOTOS). The passage of time, before an output event amoutns to may clock events taking place before te desired event happens.

```
            video !getMoviePict(M);
            videoPlayer[...] (setRate(M,0))
[]    actionPause;
            video !getMoviePict(M);
            videoPlayer[...] (setRate(M,0))
[]    actionGetMovieTime;
            data !movieTime(M);
            videoPlayer[...] (M)
[]    actionGotoTime ?targetPosition:Int;
            video !getMoviePict(setMovieTime(M,targetPosition));
            videoPlayer[...] (setMovieTime(M, targetPosition))
[]    actionSetSelection ?selection:segment;
            videoPlayer[...] (setSelection(M, selection))
[]    actionSetAllFrames;
            videoPlayer[...] (setAllFrames(M, true))
[]    actionSetVolume ?newVolumeLevel:Int;
            videoPlayer[...] (setVolume(M,newVolumeLevel))
[]    actionGetMovieVolume !volume(M);
            videoPlayer[...] (M)
[]    mcSetMovieBox?R:rct;
            videoPlayer[...] (setMovieBox(M,R))
endproc
```

Processes *playF* and *playB* have only two parameters: the flag ready and movie data M used exactly the same as in the timed model. The sycnrhonisation events with the cock are replaced with hidden events representing the non-determinstic choice between behaviour expressions that represent:

1. output of a frame (possibly the last).

2. missing a frame but continue waiting till one is displayed. Again there are two subcases depending on the falg *allFrames*(M).

If the display is not ready, the process will simply recurse.

Process *playB* is presented for the untimed model. *playF* is very similar.

```
process playB[...] (ready:Bool,M:movie):noexit:=
      [ready and (duration(M) lt movieTime(M))] ->
            video !videoFrame(M);
            data !movieTime(M);
            playB[...](false, decr(M))
[]
      [ready and (movieTime(M) eq  duration(M))] ->
            video !videoFrame(M);
            out_ok;
            data !movieTime(M);
            videoPlayer[...](M )
[]
      [ready]      -> i;
      (hide missingFrame in
            ([not(allFrames(M))] -> missingFrame;
            playB[...](ready, decr(M))
      []
            [allFrames(M)] -> missingFrame;
            playB[...](ready, M)))
[]
      [not(ready)] -> i;
            playB[...](ready, M)
[]
      actionGetMovieTime;
            data !movieTime(M);
            playB[...](ready, M)
```

```
[]
      actionGotoTime ?targetPosition:Int;
            video !getMoviePict(setMovieTime(M,targetPosition));
            videoPlayer[...] (setMovieTime(M, targetPosition))
[]
      out_ok;
      playB[...](true, M)
[]
      actionSetVolume ?newVolumeLevel:Int;
      playB[...] (ready, setVolume(M,newVolumeLevel))
[]
      actionGetMovieVolume !volume(M);
      playB[...](ready, M)
[]
      actionPlay?rt:Int [0 lt rt];
      playF[...](true, setRate(M,rt))
[]
      actionPlay?rt:Int [rt lt 0];
      playB[...] (true, setRate(M,rt))
[]
      actionPlay?rt:Int [rt eq 0];
      video !getMoviePict(M);
      videoPlayer[...] (setAllFrames(setRate(M,0),false))
[]
      actionPause;
      video !getMoviePict(M);
      videoPlayer[...] (setAllFrames(setRate(M,0),false))
[]
      mcSetMovieBox?r:rct;
      playB[...] (ready, setMovieBox(M,r))
endproc
```

7. Specification of the user interface

Specification Style

The ADC model defines more than a specification style. It is a higher level abstraction imposed on the specification language (LOTOS). It provides generic constructs which are specialised to describe the individual components of the interface software. These constructs refer to the generic assumed structure of interactive components. However there are still several remaining choices to be made as to the way the specification will be written. The main choices refer to:

- The level of detail in the description of the semantics.
- The specification style adopted for the temporal constraints applying to the interaction of each component.

For the description of the semantics no clear-cut choice was made. An instantiation of the standard operations upon received values and the local state, as defined by the abstraction and display statuses, could be enough to construct a first cut specification of the interface software. Such a specification would be purely syntactic i.e. it would simply consist of the signature of the data type. For the case study equations were given relating these operations, resulting in a more meaningful abstraction of the behaviour of the interface. Still the specifications of abstract data types are not complete. This requires a level of rigour not useful for the purposes of this study. E.g. when a point on a slider is mapped to an integer value, the specification is purely syntactic. The semantics of this mapping are not modelled.

For the description of the constraints in the dialogue no standard method is uniformly used. Two techniques (described in [4]) are used most:

- constraint oriented style of specification.
- monolithic style. In simple triggering behaviours it is more clear to simply list the allowable sequences of events for the interactor in question.

Globally, the result is a mixed resource oriented specification style [4]. This style associated with data hiding has been called object based [5]. A resource oriented style is characteristic of an implementation bias that is natural to the architectural model. More abstract specifications resulting from the composition of interactors to higher level compound ones, would include more elements of the more abstract constraint oriented style. Further investigation of the composition of interactors is currently under way.

Diagrammatic representation

Figure 4 is a graph representing the composition of interactors to model the interface. Component interactors are shown as rounded rectangles, the nodes of the graph. The larger retangle at the top represents the functional core. Connections between interactors are shown as edges connecting the interactors. Arrowheads on edges, when they exist denote a directed data flow. As mentioned elsewhere data flow between interators is directed. When both edges of the line have arroheads this means that the two connected components are receiving data from or sending data to the environement of the specification syncrhonously. The environement in this case is the user, interacting with these components through some workstation agent.

The end-points of the arcs are gates of the connected interactors. However their label is not that used locally in the interactor definition, but the identifier used for the instantiation of the interactor at the global level specification. Unconnected gates interact with the environment, and are shown as small empty rectangle boxes. If these the output gates are given a special stats. Some interactors have a single output gate on which they output the value of their display status component.

There is a direction implied in the diagram: so far the display side and the abstraction side have been distinguished. It is more accurate to distinguish between the abstraction side and the display side. So the top side of a node as it is drawn on paper, is defined to be the ***abstraction side***. The bottom side is the ***display side***. An interactor that receives data from the display side wil intepret this input with operations input() and echo() to modify the value of the abstraction and display statuses respectively. On the abstraction side input is interpreted with reeive to update abstraction status and with render to update the display status.

Some visual cues have to be added to the diagrammatic notation to show that the interactor has a display and an abstration component. This infromation can be derived from the connectivity of the node, but the diagram becomes much cleared if this information is recorded explicitly. As is discussed below, some of the interactors have one component nil, e.g. a nil abstraction component, a nil display component (there has been no example as yet of nil constraints component). The directedness of the graph is better represented if the rounded rectangles are substituted with a directed entity e.g. arches.

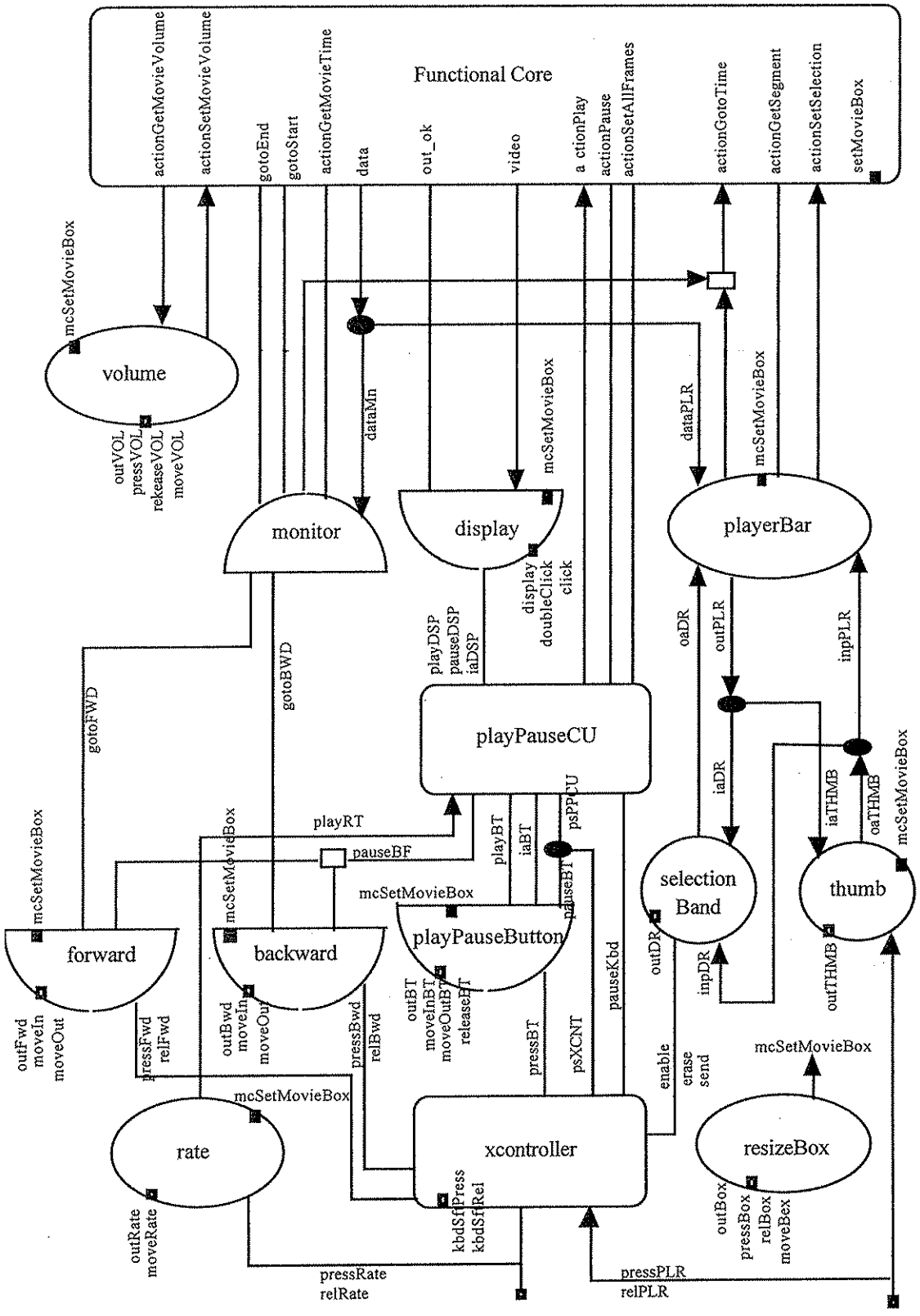

Figure 4. Diagrammatic representation of the specification structure.

8. The composition of the interface from other interactors

```
fnctCore[actionPlay, actionGetMovieTime, out_ok, actionGotoTime,
        gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames,actionSetSelection, actionSetVolume,
        ationGetMovieVolume, mcSetMovieBox, video, data, init]

|[actionPlay, actionGetMovieTime, out_ok, actionGotoTime,
gotoBeginningOfMovie, gotoEndOfMovie, actionSetAllFrames,
actionSetSelection, actionSetVolume, actionGetMovieVolume, mcSetMovieBox,
video, data]|

(demux4[data, dataPLR, dataMN]
|[dataPLR, dataMN]|
(((  (      (     playPauseACU[s, su, ab, playDSP, playBT, playRT,
                        actionPlay, pauseDSP, psPPCU, pauseKbd, pauseBF,
                        iaDSP, iaBT, optionPlay, actionSetAllFrames]
         |[s, su, ab, playDSP, pauseDSP, playBT, psPPCU, iaBT, iaDSP]|
            (     dsp[s, su, ab, click, doubleClick, display, iaDSP,
                        mcSetMovieBox, video, out_ok, playDSP, pauseDSP]
            |[s, su, ab, mcSetMovieBox]|
                  (     playPauseButton[s, su, ab, pressBT, moveInBT,
                              moveOutBT,releaseBT, outBT, mcSetMovieBox,
                              iaBT, playBT, pauseBT]
                  |[pauseBT]|  demux2[pauseBT, psPPCU, psXCNT])))
      |[s, su, ab, mcSetMovieBox]|

      (     (     (     plr[s, su, ab, inpPLR, oaDR, outPLR,
                              mcSetMovieBox, dataPLR,dataPLR,
                              actionGotoTime, actionSetSelection]
                        |[outPLR]| demux1[outPLR, iaTHMB, iaDR])
                  |[s, su, ab, iaTHMB, inpPLR]|
                  (     thumb[s, su, ab, pressPLR, moveTHMB, relPLR,
                                    outTHMB, iaTHMB, oaTHMB]
                        |[oaTHMB]| demux3[oaTHMB, inpPLR, inpDR]))
            |[s, su, ab, inpDR, iaDR, oaDR]|
                  selectionBand[s, su, ab, inpDR, outDR, iaDR, oaDR,
                                    erase, enable, send])
     |[s, su, ab, pauseBF, mcSetMovieBox]|

      (volume[s, su, ab, pressVOL, moveVOL, releaseVOL, outVOL,
                  mcSetMovieBox, actionGetMovieVolume, actionSetVolume]
      |[s, su, ab, mcSetMovieBox]|
            (     monitor[s, su, ab, option, actionGetMovieTime,
                        actionGotoTime, gotoFwd, gotoEndOfMovie, gotoBwd,
                        gotoBeginningOfMovie, dataMN]
            |[s, su, ab, gotoFwd, gotoBwd]|
                  (     pushButton[s, su, ab, pressFwd, moveInFwd,
                              moveOutFwd, relFwd, outFwd, mcSetMovieBox,
                              gotoFwd, pauseBF](fwd_arrows_icon)
                  |[s, su, ab, mcSetMovieBox]|
                        pushButton[s, su, ab, pressBwd, moveInBwd,
                              moveOutBwd, relBwd, outBwd, mcSetMovieBox,
                              gotoBwd, pauseBF](bwd_arrows_icon)))))
    |[s, su, ab, mcSetMovieBox, pressFwd, relFwd, pressBwd, relBwd,
      pressBT, psXCNT, pauseKbd, erase, send, enable, pressPLR, relPLR]|
      xcontroller[s, su, ab, kbdSftPress, kbdSftRel, pressBT, psXCNT,
                  pressPLR, relPLR, pressFwd, relFwd, pressBwd, relBwd,
                  pressRate, relRate, pauseKbd, enable, send, erase])
 |[s, su, ab, mcSetMovieBox, playRT, pressRate, relRate]|
```

```
      rate[s, su, ab, pressRate, moveRate, relRate, outRate,
            mcSetMovieBox, playRT] )
|[s, su, ab, mcSetMovieBox]|
resize[s, su, ab, pressBox, moveBox, relBox, outBox, mcSetMovieBox])
                              )
```

9. Some interactor specifications

In this section almost all interactors used in the behaviour expression above specifying the composition are discussed. Only the rate controller and the resize box are not discussed, dure to their similarity to volume and thumb respectively. However their specification can be found in the complete listing of the specification code, in the annex of this report.

The display interactor.

The display interactor maintains a status of the display only. Its data type specification is trivial (a signature only), that requires the definition of the sorts *rct* (rectangle) of the data type *graphics* and *frame* and *still* of the data type*movieData.*

```
type dspad is movieData, graphics
sorts
      disp
opns
      renderR:    disp, rct          ->    disp
      renderF:    disp, frame        ->    disp
      renderS:    disp, still        ->    disp
endtype
```

These operations are invoked by the display unit as follows:

```
process du[out,iaR, iaV](dc,ds: disp) : noexit :=
      out!dc;           du[out, iaR, iaV](dc, dc) []
      iaR?x:rct;        du[out, iaR, iaV](renderR(dc,x), ds) []
      iaV?x:frame;      du[out, iaR, iaV](renderF(dc,x), ds) []
      iaV?x:still;      du[out, iaR, iaV](renderS(dc,x), ds)
endproc (* du *)
```

The dialogue specification for interactors differs only in the constraints component, which for the display is as follows

```
process constraints[...] :noexit :=
      triggers[...]
      |[iaDsp, click, doubleClick]|
      toggle[iaDsp, doubleClick, click]
endproc

process triggers[...] :noexit :=
      click;  ps;                    triggers[...] []
      doubleClick; play;             triggers[...] []
      iaGcDsp?x:rct;                 triggers[...] []
      iaV?x:frame; out?x:disp; ok;   triggers[...] []
      iaV?x:still; out?x:disp;       triggers[...] []
      iaDsp;                         triggers[...]
endproc (*triggers *)

process toggle [iaDsp, A, B] : noexit :=
      A; iaDsp;          toggle[iaDsp, B, A]
[]    iaDsp;             toggle[iaDsp, B, A]
endproc
```

In this specification extract process *triggers*, defines locally the triggering effect of the inputs to the interactor. From these, the two inputs from the display side carry no data. On receiving them the interactor will fire a command (pause or play). Inputs from the abstraction side carry data and are those described in the display unit. Finally process toggle is used to constrain the ability of the interactor to offer the *click* or the *doubleClick* event. The interactor toggles between states in which it may be clicked so as to stop playing, or double clicked so as to start playing. Gate *iaDsp* is used to connect to *playPauseACU* that maintains statuts information for the interface i.e. whether the movie is playing or not etc.

The playBar interactor

The *playbar* is the most obvious controller component of Simple Player (vis. figure 2). It presents the user with a slider which can be used to select the current position in the movie, or as an output device to show this position. The current position is modified or presented by positioning another interactor the *thumb*. The *playBar* can also be used to select a segment of the movie together with the *selectionBand* interactor.

Player data type *plr_ad*, extends playBarType whose signature is shown below.

```
type playBarType is graphics, segmentType
sorts
      playBar
opns
      changePnt:  playBar, pnt            ->    playBar
      changeRect: playBar, rct            ->    playBar
      changeLne:  playBar, line           ->    playBar
      changeScale:playBar, Int            ->    playBar
      rect  :     playBar                 ->    rct
      point :     playBar                 ->    pnt
      line  :     playBar                 ->    line
      scale :     playBar                 ->    Int
      pntToInt:   playBar, pnt            ->    Int
      intToPnt:   playBar, Int            ->    pnt
      segToLne:   playBar, segment  ->    line
      lneToSeg:   playBar, line           ->    segment
eqns
(* equations ommitted *)
endtype
```

A play bar is characterised by four sorts, a selected point, the current rectangle, a line defined by the two points that enclose and the selected segment, and a scale. Operations are defined for setting and inquiring the values of these sorts. The operations *intToPnt*, *segToLne*, *lneToSeg* and *pntToInt*, model the most important function of the interactor, which to map integer values representing time in the movie, to points on the slider rectangle. Only the signature for these operations is specified. A complete specification of the translation between sorts is outside the scope of this exercise. *plr_ad* is now defined as follows:

```
type plr_ad is playBarType
sorts playBarData
opns
      inputM:     pnt, playBar, playBarData     ->    playBarData
      inputMK:    line, playBar, playBarData    ->    playBarData
      echoM:      pnt, playBar, playBarData     ->    playBar
      echoMK:     line, playBar, playBarData    ->    playBar
      renderS:    playBar, segment        ->    playBar
      renderT:    playBar, Int            ->    playBar
      renderRct:  playBar, rct            ->    playBar
      receiveS:   playBarData, segment    ->    playBarData
      receiveT:   playBarData, Int        ->    playBarData
```

```
      receiveRct: playBarData, rct          ->    playBarData
      resultT:    playBarData               ->    Int
      resultS:    playBarData               ->    segment
eqns
forall r:rct,p:pnt,ln:line,pb:playBar,ts:playBarData,s:segment,t:Int
ofsort Int
      resultT(inputM(p,pb,ts)) = pntToInt(pb,p);
      resultT(inputMK(ln,pb,ts))=resultT(ts);
      resultT(receiveS(ts,s))=resultT(ts);
      resultT(receiveT(ts,t))=t;
      resultT(receiveRct(ts,r))=resultT(ts);
ofSort segment
      resultS(inputM(p,pb,ts)) = resultS(ts);
      resultS(inputMK(ln,pb,ts))= lneToSeg(pb, ln);
      resultS(receiveS(ts,s))=s;
      resultS(receiveT(ts,t))=resultS(ts);
      resultS(receiveRct(ts,r))=resultS(ts);
ofSort playBar
      echoM(p, pb, ts) = changePnt(pb, p);
      echoMK(ln,pb,ts) = changeLne(pb, ln);
      renderS(pb,s) = changeLne(pb, segToLne(pb,s));
      renderT(pb,t) = changePnt(pb, intToPnt(pb, t));
      renderRct(pb,r) = changeRect(pb,r);
endtype
```

PlayBar communicates to these other interactors. It maintains its display, which it updates to reflect positioning information derived either from communication with the functional core, or via interactions with the thumb.

```
process adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            (a:playBarData, dc, ds: playbar) : noexit :=
  inp_m?x:pnt;          adu[...](inputM(x,ds,a),echoM(x,ds,a),ds) []
  inp_mk?x:line;        adu[...](inputMK(x,ds,a),echoMK(x,ds,a),ds) []
  oa_t!resultT(a);      adu[...](a, dc, ds) []
  oa_s!resultS(a);      adu[...](a, dc, ds) []
  out!dc;               adu[...](a, dc, dc) []
  iaRct?x:rct;          adu[...](receiveRct(a,x),renderRct(dc,x), ds)[]
  iaSg?x:segment;       adu[...](receiveS(a,x),renderS(dc,x), ds)[]
  iaNt?x:Int;           adu[...](receiveT(a,x),renderT(dc,x), ds)
endproc
```

Interactor *plr* maintains a single abstraction *a*, but outputs two results, two interpretations of this abstraction on gates *oa_t* and *oa_s* respectively. Similarly it may receive two different types of input which may be a single point on gate *inp_m*, or a pair of points forming a segment in gate *inp_mk*. It has only one output gate where its display status is output. It may receive three types of input from the abstraction side that are not interpreted with respect to the display. The constraints for this behaviour are described in process *constraints* below.

```
process constraints [inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s] :
noexit :=
      iaRct?x:rct;                    constraints[...]
[]    iaSg?x:segment;                 constraints[...]
[]    iaNt?x:Int; out?y:playbar;      constraints[...]
[]    out?x:playbar;                  constraints[...]
[]    inp_m?x:pnt; oa_t?y:int;        constraints[...]
[]    inp_mk?x:line; oa_s?y:segment;constraints[...]
endproc
```

The only dialogue constraints imposed by the above are that the interactor will output its display status after receiving a new value for the position in the movie from gate *iaNt*, and it will immediately relay its interpretation of any input of point or line from the display side to the abstraction side.

The technique for the specification of the dialogue constraints is in this case very simple. Process constraints constrains all gates of the interactor apart from start, suspend and abort gates (cf. [1]). Thus they should all be in its alphabet. For gates where no triggering behaviour is described, the event is offered and process constraints simply recurses. Where the event triggers another event, this is denoted with simply prefixing the action enabled before the recursive clause.

The thumb

The thumb (or indicator) is an interactor on its own right, although it works in conjunction with the player bar described above. It may receive position information from the user, such as a mouse press, or movement on the player bar. It will position itself at the point of the last mouse press, or if the movie is playing it will indicate the current position in the movie. It acts as an input device for selecting a particular position in the movie and as an output device that allows the user to observe the current position in the movie.

The data type *thumb_ad* which is used by the interactor, uses *playBarType* described above, and the *thumbDisplayType* below.

```
type thumbDisplayType is playBarType
sorts
      thumb_dsp
opns
      moveThumb:  thumb_dsp, pnt          ->    thumb_dsp
      2Dto1Dpnt:  thumb_dsp, pnt          ->    pnt
eqns
forall td: thumb_dsp, pnt1, pnt2: pnt
ofsort thumb_dsp
      moveThumb(moveThumb(td,pnt1),pnt2)=moveThumb(td, pnt2);
endtype
```

This data type, simply states that the icon of the thumb can be moved, and that the thumb has no memory of previous positions. The interactor may receive data from the mouse press, or move or release. Interpretation of user input is trivial, because the abstraction is simply the value of the point last input. This value is relayed to the play bar, so the result operations is simply an identity function. The only interesting operation is to extract position information from the current output of the player bar.

```
type thumb_ad is Graphics, playBarType, thumbDisplayType
opns
      inputPress   :     pnt, thumb_dsp, pnt     ->    pnt
      inputMove    :     pnt, thumb_dsp, pnt     ->    pnt
      inputRelease:      pnt, thumb_dsp, pnt     ->    pnt
      echoPress    :     pnt, thumb_dsp, pnt     ->    thumb_dsp
      echoMove     :     pnt, thumb_dsp, pnt     ->    thumb_dsp
      echoRelease  :     pnt, thumb_dsp, pnt     ->    thumb_dsp
      render       :     thumb_dsp, playBar      ->    thumb_dsp
      receive      :     pnt, playBar            ->    pnt
      result       :     pnt                     ->    pnt
eqns
forall a,p:pnt, d:thumb_dsp, pb:playBar
ofsort  pnt
      inputPress(p, d, a) =  2Dto1Dpnt(d,p);
      inputMove(p, d, a) = 2Dto1Dpnt(d,p);
      inputRelease(p, d, a) = 2Dto1Dpnt(d,p);
ofsort thumb_dsp
      echoPress(p, d, a) = moveThumb(d,p);
      echoRelease(p, d, a) = moveThumb(d,p);
      echoMove(p, d, a) = moveThumb(d,p);
      render(d, pb) = moveThumb(d,point(pb));
ofsort pnt
```

```
      receive(a, pb) = point(pb);
      result(a) = a;
endtype
```

The constraints for the thumb interactor are:

```
process constraints[press, move, release, out, ia, oa] : noexit :=
      inp[press, move, release]
            |[press, move, release]|
      trigger[press, move, release, out, oa, ia]
endproc

process trigger[press, move, release, out, oa, ia]: noexit :=
      (choice X in [press,move,release]
            []X?y:pnt; out?z:thumb_dsp; oa?q:pnt;
                  trigger[press, move, release, out, oa, ia])
      []
      (ia?x:playBar;
      out?z:thumb_dsp; trigger[press, move, release, out, oa, ia])
endproc(* trigger *)

process inp[press, move, release]: noexit :=
      press?x:pnt;
      (repeat[move]
            [>
      release ?x:pnt;
      thumbDial[press, move, release])
endproc

process repeat[inp]:noexit :=
      inp?x:pnt; repeat[inp]
endproc
```

The selection band

The selection band interactor controls user input to the player bar, constructing an abstraction that represents the current selection as a simple line segment. This is represented as a band on the rectangle area of the play bar. Its result is sent to the *playBar*. The *playbar*, on receiving such input will interpret it to the current selection in movie time coordinates. Its display modifies the output of the player bar.

The selection band maintains a line as its local state, and a slider graphic entity (the play bar) for its display status. It receives points as input, or a signal *erase* which is used to deselect a line. Erase is an example of an input event carrying no data. It is though modelled within the *adu* process, as it evokes interpretation functions *inputEr*, and *echoEr*.

```
process adu[inpPnt, out, ia, oa, erase](a:line,pc,ps:playBar):noexit :=
      oa!a;              adu[inpPnt, out, ia, oa, erase] (a,pc,ps)
[]    out!pc;            adu[inpPnt, out, ia, oa, erase] (a,pc,pc)
[]    ia?x:playBar;      adu[inpPnt, out, ia, oa, erase]
                               (receivePB(a,x),renderPB(pc,x),ps)
[]    inpPnt?x:pnt;      adu[inpPnt, out, ia, oa, erase]
                               (inputPnt(x,a),echo(x,ps,a),ps)
[]    erase;             adu[inpPnt, out, ia, oa, erase]
                               (inputEr(a),echoEr(pc, a),ps)
endproc
```

The interactor will respond with an echo to input events from the display side (*inpPnt* and *erase*). However, the dialogue is constrained as it needs to be enabled by a signal on gate *enable*. It will be interrupted by the signal *send*, which makes the interactor send the line segment it has constructed to the player bar with event *oa?x:line*. An *erase* event also causes this communication although in this case a *nil* line is sent.

```
process constraints[inpPnt, out, ia, oa, erase, enable, send] : noexit :=
      enable; (operation[inpPnt, out, ia, oa, erase, send]
                  [> send; oa?x:line; constraints[...])
[]    ia?x:playBar;                          constraints[...]
[]    out?x:playBar;                         constraints[...]
[]    erase; out?x:playBar;oa?x:line;        constraints[...]
endproc  (* constraints *)

process operation[inpPnt, out, ia, oa, erase, send] : noexit :=
      inpPnt?x:pnt; out?x:playBar; operation[...]
[]    out?x:playBar; operation[...]
endproc (* operation *)
```

The abstract data type used by the selection band is listed below. Notice how there are two input and echo functions, used correspondingly when a point is input, or when an erase signal is input. The equations make the specification more meaningful by relating the standard operations, to custom operations of *selection_abs* and *band_dsp* data types. Further equations are specified to the effect that the play bar has no memory after the player bar has been reset (by input from the *playerBar* interactor).

```
type selection_ad is selection_abs, band_dsp
opns
      inputPnt:    pnt, line              ->    line
      inputEr:     line                   ->    line
      echo:        pnt, playBar, line     ->    playBar
      echoEr:      playBar, line          ->    playBar
      receivePB:   line, playBar          ->    line
      renderPB:    playBar, playBar       ->    playBar
eqns
forall p:pnt, a:line, pb,pb1,pb2:playBar
ofsort  line
      inputPnt(p,a) = getPoint(a,p);
      inputEr(a)=nil;
      receivePB(receivePB(a,pb1),pb2)=receivePB(a,pb2);
ofsort playBar
      renderPB(pb1,pb2) = pb2;
      echo(p,pb,a)=hilite(pb,getPoint(a,p));
      echoEr(pb,a)=dehilite(pb);
endtype
```

Data types *selection_abs* amd *band_dsp* hide from the *selection_ad* data type the particular complexities of the data types they handle. In fact, they can be thought of as implementing the operations of *selection_ad*. The specifications of them included below, are more involved than *selection_ad*, they are case specific, and cannot be derived from the model. In general, throughout this case study, the data type particular to the interactor, can be thought of as an instantiation of the general signature for the specification of the data handling aspects of the ADC model (cf. [1]). This data type is made more meaningful to the context it is applied in by reusing such case specific data types. The specification style for these case specific data types is not constrained.

Below, the abstraction maintains a pair of points that define the selection. It keeps the leftmost and rightmost points input. When dragging the thumb across the selection band, the user inputs a series of points. If the first point (the start of the selection) is at the left of the line it is made its left boundary. If it is to the right of the line it becomes the right boundary. How this decision is made (e.g. by comparing with the middle of the line), is not modelled below. It is enough to represent the conditional behaviour with the two Boolean functions *leftOf* and *rightOf*. comparing a point to a line.

```
type selection_abs is graphics, Boolean
opns
      nil:                          ->    line
      reset:              line      ->    line
```

```
      setStart, setEnd: line, pnt    ->    line
      getPoint:         line, pnt    ->    line
      startSel, endSel:       line         ->    pnt
      isNil:                  line         ->    Bool
      _ leftOf _:       pnt, line    ->    Bool
      _ rightOf _:            pnt, line    ->    Bool
eqns
forall d,d1,d2:line, p,p1,p2:pnt
ofsort Bool
      isNil(nil)=true;
      isNil(setStart(d,p))=false;
      isNil(setEnd(d,p))=false;
      isNil(getPoint(d,p))=false;
ofsort pnt
      startSel(setStart(d,p))=p;
      endSel(setStart(d,p))=endSel(d);
      startSel(setEnd(d,p))=startSel(d);
      endSel(setEnd(d,p))=p;
ofsort line
      setStart(setStart(d,p1),p2)=setStart(d,p2);
      setEnd(setEnd(d,p1),p2)=setEnd(d,p2);
      setStart(setEnd(d,p1),p2)=setEnd(setStart(d,p2),p1);
      reset(d)=nil;
      getPoint(nil, p) = setEnd(setStart(nil, p),p);
      not(isNil(d)) and (startSel(d) leftOf d) =>
            getPoint(d,p) = setStart(d,p);
      not(isNil(d)) and (startSel(d) rightOf d) =>
            getPoint(d,p) = setEnd(d,p);
endtype
```

As for the display of operations for the selection band, all that is said, is that given the current display of the player bar, a line segment on it can be highlighted, or de-highlighted when the selection is erased.

```
type band_dsp is playBarType
opns
      hilite:     playBar, line     ->    playBar
      dehilite:   playBar           ->    playBar
eqns
forall pb:playBar, l,m:line
ofsort playBar
      hilite(hilite(pb,l),m)=hilite(pb,m);
      hilite(dehilite(pb),m)=hilite(pb,m);
      dehilite(hilite(pb,m))=pb;
```

endtype

The dialogue is describe din the constraints component as follows.

```
process constraints[...] : noexit :=
      (enable; (operation[inpPnt, out, ia, oa, erase, send]
                  [> send; oa?x:line;     constraints[...]))
[]    ia?x:playBar;                       constraints[...]
[]    out?x:playBar;                      constraints[...]
[]    erase; out?x:playBar;oa?x:line;     constraints[...]
endproc

process operation[inpPnt, out, ia, oa, erase, send] : noexit :=
      inpPnt?x:pnt; out?x:playBar;  operation[...]
[]    out?x:playBar;                operation[...]
endproc
```

The Play/Pause Button

This interactor is typical of the complex behaviours that are packaged in the movie controller components. This button, may be used to start playing. Once this is done, it turns itself into a pause button. Its presentation changes according to its function. The play/pause button is sensitive also to the user dragging the mouse outside its own active area. If the user has pressed the play button and drags the mouse outside a region surrounding the icon, the movie will stop. The button will be de-highlighted although it will not change its icon.

When a play or pause command is issued by another interactor, the icon and function of the play/pause button changes accordingly. This is emulated in the interactor specification by a signal, which, as can be seen in the overall specification diagram, is coming from a controller interactor *playPauseACU*. The latter is specifically constructed to coordinate all sources of play or pause events. This contributes to the modularity of the specification: the play pause button should not be aware of all other interactors that might indirectly affect its behaviour by issuing a play or pause command.

The play pause button is a display only interactor component, in that it does not maintain a local abstraction state. Its state as playing or not, is captured as a dialogue state by the process algebra specification of the dialogue constraints in the controller component. The display unit of the interactor is responsible simply for updating the display to signals received from other interactors (*iaGC, ia*) or directly form the user (*press, moveIn, moveOut, release*).

```
process du[press, moveIn, moveOut, release,  out, iaGC, ia]
                                    (dc,ds: twoStateButton_dsp) : noexit
:=
      out!dc; du[...] (dc, dc) []
      iaGC?x:rct; du[...] (renderR(dc,x), ds) []
      ia; du[...] (renderS(dc), ds) []
      press; du[...] (echoPr(dc), ds) []
      moveIn;  du[...] (echoMovIn(dc), ds) []
      moveOut; du[...] (echoMovOut(dc), ds) []
      release; du[...] (echoRel(ds), ds)
endproc.
```

In this display-only interactor, the local state is captured by the parameters *dc* and *ds*, representing the next and current display state of the interactor. The operations upon them are specified by the data type *playPause_d* as follows.

```
type playPause_d is twoStateButton
opns
      echoPr      :     twoStateButton_dsp       ->    twoStateButton_dsp
      echoRel     :     twoStateButton_dsp       ->    twoStateButton_dsp
      echoMovIn   :     twoStateButton_dsp       ->    twoStateButton_dsp
      echoMovOut  :     twoStateButton_dsp       ->    twoStateButton_dsp
      renderR     :     twoStateButton_dsp, rct  ->    twoStateButton_dsp
      renderS     :     twoStateButton_dsp        ->   twoStateButton_dsp
eqns
forall p:pnt, ppd:twoStateButton_dsp, r:rct
ofsort twoStateButton_dsp
      echoPr(ppd)       = setHilite(ppd, onH);
      echoMovIn(ppd) = setHilite(ppd, onH);
      echoMovOut(ppd) = setHilite(ppd, offH); ;
      echoRel(ppd) = setHilite(switch(ppd),offH);
      renderR(ppd,r)    = setRect(ppd,r);
      renderS(ppd)      = switch(ppd);
endtype
```

As usual this data type uses a more concrete specification *twoStateButton* whose signature is included below.

```
type twoStateButton is graphics
sorts
      twoStateButton_dsp
opns
      mkButtonDsp:        icon, rct, hilite          ->    twoStateButton_dsp
      getIcon:            twoStateButton_dsp         ->    icon
      getHilite:          twoStateButton_dsp         ->    hilite
      getRect:            twoStateButton_dsp         ->    rct
      setIcon:            twoStateButton_dsp, icon->    twoStateButton_dsp
      setRect:            twoStateButton_dsp, rct ->    twoStateButton_dsp
      setHilite:          twoStateButton_dsp, hilite -> twoStateButton_dsp
      switch:             twoStateButton_dsp         ->    twoStateButton_dsp
      playIcon, pauseIcon:                           ->    icon
      onH, offH:                                     ->    hilite
eqns
(* equations are ommitted here *)
entype
```

As mentioned above, the dialogue structure is quite complex. The process constraints of the controller component which describes this dialogue is listed below; it specified using the constraint oriented style. It is formed as the parallel composition of four processes, putting partial constraints on their synchronisation gates. Individual processes represent different types of constraints. Input-output constraints, e.g. firing an echo are described in process *inp_out*. Triggering behaviour is described in triggers. Process *inp* expresses the low level constraints in the input side only e.g. the user cannot release without having pressed the mouse first. Finally, *toggle* represents the two dialogue states discussed previously: in one the interactor triggers a pause command in the other a play command. The interactor will toggle between these two states with an event *ia* received from external sources.

```
process constraints[...] : noexit :=
((inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
      |[press, moveIn, moveOut, release]|
inp[press, moveIn, moveOut, release])
      |[press, moveIn, moveOut, play, pause]|
triggers[press, moveIn, moveOut, play, pause])
      |[play, ia, pause]|
toggle[play, ia, pause]
endproc

process triggers[...]: noexit :=
      (choice G1 in [press, moveIn, moveOut]
            [] G1; (choice G2 in [play, pause] [] G2;
                  triggers[...]))
endproc

process inp_out[...]: noexit :=
      (choice G in [press, moveIn, moveOut, release, ia] [] G;
                              out?x:twoStateButton_dsp;  inp_out[...])
      [] iaGC?x:rct; out?x:twoStateButton_dsp;
         inp_out[...]
      [] (choice G2 in [play, pause] [] G2; ia;
         inp_out[...])
endproc

process inp[...]: noexit :=
      press; (repeat2[moveIn, moveOut][>
            release; inp[...])
endproc
```

```
process repeat2[moveIn, moveOut]:noexit :=
      moveOut; moveIn; repeat2[moveIn, moveOut]
endproc

process toggle[A, ia, B]: noexit :=
      A; ia; toggle[B, ia, A]
      []
      ia; toggle[B, ia, A]
endproc
```

The playPause ACU interactor

So far, only interactors that are visible to the user have been discussed. Indeed, at the commencement of this exercise it was foreseen that only the interactors visible to the user would be needed to model the Simple Player™ interface. The need for other interaction objects, arises from the fact, that individual interactor behaviours affect each other. Rather than having such dependencies handled by the affected interactors, they are assigned to separate interactors for reasons of modularity. Interactor playPauseACU is such an interactor whose responsibility is to receive play and pause commands from their various possible sources (the display interactor, the play pause button, the rate controller etc.) and issue the actual command to the functional core. It will inform interactors of its state changes, i.e. if it is about to send a play or a pause command.

Interactors that are not visible to the user do not have a display status, or operations upon it like render and echo operations. If they have no abstraction state either then the abstraction display unit can be dispensed with altogether, and the interactor will be reduced to its controller component.

In the case of playPauseACU, the interactor maintains a local state, described by the current rate. The abstraction unit, will handle input events which are interpreted via receive operations (i.e. operations that do not use the current display state as an operand). In fact the receive operation in this case is an identity function, so the abstraction unit can be written simply as:

```
process au[ia, oa](r:int):noexit :=
      ia?x:int;    au[ia, oa](x)
[]    oa!r;        au[ia, oa](r)
endproc
```

The dialogue constraints upon this simple unit, and upon the signal events that are relevant to the controller component only, are quite more complex. A command play may be received from a number of sources: the display interactor, the play/pause button, the rate controller. The command is immediately passed on to the application via gate *actionPlay*. If though an option keyboard modifier is detected, the attribute *playAllFrames* of the movie will be set to true with the action *setAll*.

All the affected interactors are informed when *actionPlay* has been accepted by the application. In this case it is the display and the play/Pause button interactors. The order of these notification events is not significant (therefore the choice of orders).

While in this dialogue state, i.e. after having issued the command to play, the interactor may still receive requests to change the rate of playing of the movie, originating from the rate controller. This can be repeated, as is shown with process *changesOfRate*, until a pause command is issued from one of the interactors. The actual command issued to the application is *actionPlay* with a value 0 for the rate.

```
process constraints[...]:noexit :=
(    (playDSP; exit [] playBT; exit []  playRT?x:int; exit [] option;
            playBT; setAll; exit) >>
```

```
      (actionPlay?y:int;     (iaBT; iaDSP; exit [] iaDSP; iaBT; exit) >>
            (      changesOfRate[playRT, actionPlay]
            [>
                  (choice B in [pauseDSP, pauseBT, pauseKbd, pauseBF]
                        [] B; actionPause;
                        (iaBT; iaDSP; exit [] iaDSP; iaBT; exit))))
 []   pauseBF; exit)
>>
constraints[...]
endproc

process changesOfRate[p,a] : noexit :=
      p?x:int; a?y:int; changesOfRate[p,a]
endproc
```

The volume interactor

The volume interactor is characterised by two distinct presentations. When dormant it is shown as a speaker icon. When activated a slider pops up with which the user can manipulate the volume of the sound for the movie that is playing. The interactor similar to other slider components will map points to integers and vice versa. The data type handled by the interactor is volume_ad below, which uses the data type *popUpSlider* whose signature only is included here.

```
type popUpSlider is graphics, Integer
sorts
      volumeBar
opns
      popUpSlider:              volumeBar    ->    volumeBar
      chIconAndSlider:  volumeBar, pnt       ->    volumeBar
      popDownSlider:            volumeBar    ->    volumeBar
      changeRect:       volumeBar, rct       ->    volumeBar
      pntToInt:         volumeBar, pnt       ->    Int
      IntToBar:         volumeBar, Int       ->    volumeBar
eqns
(* equations omitted here *)
endtype

type volume_ad is popUpSlider
opns
      inputPr:    volumeBar, Int             ->    Int
      echoPr:     volumeBar, Int             ->    volumeBar
      inputMov:   pnt, volumeBar,Int         ->    Int
      echoMov:    pnt, volumeBar, Int        ->    volumeBar
      inputRel:   volumeBar,Int              ->    Int
      echoRel:    volumeBar, Int             ->    volumeBar
      renderRV:   volumeBar, rct             ->    volumeBar
      receiveRV:  Int, rct                   ->    Int
      renderV:    volumeBar, Int             ->    volumeBar
      receiveV:   Int, Int                   ->    Int
      result:     Int                        ->    Int

eqns
forall r:rct, p:pnt, v, n:Int, vb:volumeBar
ofsort Int
      result(v) = v;
      inputPr(vb,v) = v;
      inputMov(p,vb,v) = pntToInt(vb,p);
      inputRel(vb,v) = v;
      receiveRV(v,r)=v;
      receiveV(v,n)=n;
ofSort volumeBar
```

```
      echoPr(vb, v) = popUpSlider(vb);
      echoMov(p, vb, v) = chIconAndSlider(vb,p);
      echoRel(vb, v) = popDownSlider(vb);
      renderRV(vb,r) = changeRect(vb,r);
      renderV(vb,v) = IntToBar(vb,v);
endtype
```

This data type is handled by the process *adu* as follows:

```
process adu[press, move, release,  out, iaR, iaV, oa]
            (a:Int,dc,ds:volumeBar) : noexit :=
      out!dc;           adu[...] (a, dc, dc)
[]    iaR?x:rct;        adu[...] (receiveRV(a,x), renderRV(dc,x), ds)
[]    iaV?x:Int;        adu[...] (receiveV(a,x), renderV(dc,x), ds)
[]    press;            adu[...] (inputPr(ds,a), echoPr(ds,a), ds)
[]    move?x:pnt;       adu[...] (inputMov(x,dc,a), echoMov(x,dc,a), ds)
[]    release;          adu[...] (inputRel(ds,a), echoRel(ds,a), ds)
[]    oa!result(a);     adu[...] (a, echoRel(ds,a), ds)
endproc
```

The controller component for the volume interactor is

```
process constraints[press, move, release, out, iaR, iaV, oa] :noexit:=
      triggers[press, move, release, out, iaR, iaV, oa]
            |[press, move, release]|
      inp_control[press, move, release]
endproc
```

The component processes are very similar to other interactors and need not be discussed further.

The step forward and back buttons

These are simple push buttons. They do not maintain a local abstraction status, so they do not need to interpret user input. However, as with the play/pause button, they are sensitive to the user dragging the mouse out of their active region. The two buttons differ only in that they are instantiated with two different icons for a display status, and in their connectivity with other interactors. Only the forward button is presented below.

Already, it was mentioned that the buttons do not have an abstraction status. The specification of their display only unit, uses in fact the data type *twoStateButton_dsp* defined for the play/pause button. The two displays are different only with respect to which icon is presented to the user.

```
process du[...](dc,ds: twoStateButton_dsp) : noexit :=
      out!dc;     du[...] (dc, dc) []
      iaGC?x:rct; du[...] (renderR(dc,x), ds) []
      press;      du[...] (echoPr(dc), ds) []
      moveIn;     du[...] (echoMovIn(dc), ds) []
      moveOut;    du[...] (echoMovOut(dc), ds) []
      release;    du[...] (echoRel(ds), ds)
endproc
```

Its dialogue, is very similar to the play/pause button as well. Again it is defined as a composition of constraints on the input side, or input-output constraints and the trigger behaviour of the interactor.

```
process constraints[press, moveIn, moveOut, release, out, iaGC, goto,
pause] : noexit :=
      (inp_out[press, moveIn, moveOut, release, out, iaGC]
            |[press, moveIn, moveOut, release]|
      inp[press, moveIn, moveOut, release, pause])
            |[press, moveIn, release, moveOut, iaGC, out]|
```

```
      triggers[press, moveIn, moveOut, release, goto, iaGC, out]
endproc
```

For brevity only the triggering behaviour is described.

```
process triggers[press,moveIn,moveOut,release,goto,iaGC,out]:noexit :=
      press;       goto;        triggers[...]
[]    moveIn;      goto;        triggers[...]
[]    moveOut;                  triggers[...]
[]    release;                  triggers[...]
[]    iaGC?x:rct;               triggers[...]
[]    out?x:twoStateButton_dsp;      triggers[...]
endproc
```

The last four clauses ensure that the trigger process constrains the interactor to perform a *goto* after a press or a moveIn, prior to accepting any of the last four events. Nevertheless *inp_out* does not offer *pause*, and *pause* is not in the set of synchronisation gates. Thus it is not affected by this process and is only fired when process *inp* determines.

The monitor interactor

Another interactor that does not have its own display is the monitor interactor. It is not immediately obvious that such an interactor is needed to mediate between the forward and backward step buttons and the functional core. It is necessary because, the two buttons are little more than simple command buttons firing a command and do not maintain position information about the movie. *Monitor* records where the movie is, and will move forward or backward one time unit. It will also move to the start or end of the movie if it detects an option key modifier event. Its abstraction unit maintains only some information of sort *montr_abs*, defined in data type *montr_ad*. This uses*timeInfo* whose signature only is included.

```
type timeInfo is Integer
sorts
      montr_abs
opns
      now:          montr_abs          ->    int
      dt:           montr_abs          ->    int
      end:          montr_abs          ->    int
      setNow:       montr_abs, int     ->    montr_abs
      setDt:        montr_abs, int     ->    montr_abs
      setEnd:       montr_abs, int     ->    montr_abs
eqns (*equations are omitted*)

type montr_ad is timeInfo
opns
      receive:      montr_abs, int     ->    montr_abs
      receiveD:     montr_abs, int     ->    montr_abs
      jumpF:        montr_abs          ->    montr_abs
      jumpB:        montr_abs          ->    montr_abs
      inputF:       montr_abs          ->    montr_abs
      inputB:       montr_abs          ->    montr_abs
      result:       montr_abs          ->    int
eqns forall t:Int, m:montr_abs
ofsort Int
      result(m) = now(m)+dt(m);
ofsort montr_abs
      receiveD(m,t) = setEnd(m,t);
      receive(m,t)= setNow(m,t);
      jumpF(m) = setNow(setDt(m,0),end(m));
      jumpB(m) =  setNow(setDt(m,0),0);
      (now(m) lt end(m))=>     inputF(m) = setDt(m,s(0));
      (0 lt now(m))=>          inputB(m) = setDt(m,p(0));
```

```
      (now(m) eq end(m)) =>    inputF(m) = m;
      (0 eq now(m)) =>         inputB(m) =m;
endtype
```

When a signal *fwd*, or *bwd* is received from the button interactors, the monitor will set an increment appropriately. This increment is added to the present time of the movie when the interactor instructs the functional core to go to the specific location.

```
process au[oa_get, oa_goto, fwd, bwd, ia](theTime:montr_abs): noexit :=
      oa_get;            au[oa_get, oa_goto, fwd, bwd, ia](theTime)
[]    ia?x:int;    au[oa_get, oa_goto, fwd, bwd, ia](receive(theTime,x))
[]    oa_goto !result(theTime);au[oa_get, oa_goto, fwd, bwd, ia](theTime)
[]    fwd;         au[oa_get, oa_goto, fwd, bwd, ia](inputF(theTime))
[]    bwd;         au[oa_get, oa_goto, fwd, bwd, ia](inputB(theTime))
```

endproc

However, the interactor may send signal only events to the functional core, instructing it to go to the start (gate *jB*) or the end (gate *jF*) of the movie. This happens if a keyboard modifier option has been pressed before the buttons are pressed.

```
process constraints[...]: noexit :=
      (choice x in [fwd, bwd] [] x;
                  oa_get; ia?z:int; oa_goto?y:int; constraints[...])
      [] option;  (     fwd; jF; constraints[...]
                  []    bwd; jB; constraints[...])
endproc
```

XController. Lexical Level interaction.

The XController is the odd interactor that stands out from the others. It does not have a display or an abstraction status, so its role is only as a dialogue controller. It was introduced to maintain the modularity of the controlled interactors, by concentrating in one place all the relevant constraints on event orderings for them.

It also stands out from other interactors, in that it really aims to distinguish between different orderings of very low level events: pressing and releasing keyboard modifiers, pressing and releasing the mouse etc. Of course, it is preferable to abstract away from such issues even in a reverse engineering example. In this case though, various orderings of these low level events result in different commands being issued to the functional core. While of a low abstraction level, the dialogues described in XController fall within the scope of the specification exercise, as they concern dialogue constraints and not the formation of input tokens.

Possibly due to this low level of abstraction, the most natural way to describe the dialogue would have been with state transitions. At higher level of abstraction, constraints work better. The state oriented style of LOTOS specification was used, which effectively treats each process as a state of a state transition network. Action prefix operators may be thought of as transitions to sub-states.

```
process constraints[...]: noexit :=
 (    kbdSftPress;  kbdSftMode[...]
[]    pressPLR?x:pnt; erase;  (exit []
                        kbdSftPress; enable;
                              (kbdSftRel; send; exit []
                              relPLR?x:pnt; send; kbdSftMode[...]))
[]    pressRate; kbdCtrlMode[...]
[]    (choice x in [pressBT, pauseBT, pressFwd, relFwd, pressBwd, relBwd]
[]x; exit) )
>> constraints[...]
endproc
```

```
process kbdSftMode[...]:noexit :=

      kbdSftRel; exit
[]    pressBT; enable;
            (kbdSftRel; send; exit
            []
            pauseBT; send;
            kbdSftMode[...])
[]    pressPLR?x:pnt; enable;
            (relPLR?x:pnt; send; kbdSftMode[...]
            []
            kbdSftRel; send; exit)
[]    pressFwd; enable; relFwd; send; exit
[]    pressBwd; enable; relBwd; send; exit
[]    pressRate; enable; (choice X in [kbdSftRel, pauseBT]
            [] X; send; kbdCtrlMode[...])
>>    constraints[...]
endproc  (* kbdSftMode *)

process kbdCtrlMode[...]:noexit :=
      relRate; constraints[...]
[]    kbdSftPress; enable; (pauseBT; send;
                  kbdSftMode[...]
                  []
                  kbdSftRel; send; pauseKbd; kbdCtrlMode[...])
endproc
```

Connecting interactors

For the purposes of this exercise, connections between interactors were quite standard. Apart form the LOTOS multi-way synchronisation which was used for data flow, or even for synchronisation only, some simple connectives turned out to be useful. For example, two interactors need to be informed of the current display of the play bar: *thumb* and *selectionBand*. The synchronisation of both with the functional core is not the desired behaviour. In the case study the connective *demux1* was introduced. This simply describes how a value received at one gate is relayed at both of two others.

```
process demux1[outPLR, iaTHMB, iaDR]: noexit :=
outPLR?x:playBar;
      (iaTHMB?y:playBar[x=y]; iaDR?z:playBar[z=x];     demux1[...]
[]    iaDR?z:playBar[z=x];    iaTHMB?y:playBar[x=y];  demux1[...])
endproc
```

Two interactors may be alternative sources of an event. For example, either of the *monitor* or the *playBar* may instruct the functional core to go to a particular time in the movie. Both interactors then need to synchronise with the functional core, but they should not be synchronised between them. Thus while *actionGotoTime* belongs to the set of synchronisation gates for the functional core it should not belong to any set of synchronisation gates that would constrain both *monitor* and *playBar*. In the following section it is attempted to provide a general classification of these connections.

These connectives could themselves be thought of as ADC interactors, (in fact they would be abstraction only interactors). Specifying start, stop etc. is superfluous, since they are only needed when the interactors they communicate to are active. In this example all interactors syncrhonise over their standard control gates (start, stop etc.) so the connectives would only operate when the interactors they syncrhonize with do so. Thus the simpler option shown above was preferred over their specification as proper interactors.

10. Improvements on the ADC Model

In the course of the specification exercise, modifications were prompted to the definitions of the interactor model. They refer to special cases of the ADC model, reconsideration of the standard control behaviours (start, suspend and abort), and the scope of local state components. The latter raises questions regarding the initialisation of the state variables of an interactor.

Signal (no data) events

The original definition of the ADC model [1] stipulates that all events over the gates of the ADU carry data, and further that the alphabet of the controller consists in the alphabet of the ADU plus the control events *start*, *suspend* and *abort*. Two more classes of events not accounted for in the original definition of the model are introduced. Both carry no data, and for convenience they are referred to as signal events[2]. These are:

1. events that carry no data but are in the alphabet of the ADU

2. controller events other than start, suspend and abort.

An example of the former can be seen in the specification of the volume interactor. In the *adu* process a mouse press event carries no data, but invokes the input and echo functions for a press event. These have the effect of popping up the volume slider:

```
press;      adu[...] (inputPr(ds,a), echoPr(ds,a), ds)
```

As an example of the controller behaviour not related to start, abort, suspend of the operation, consider the process *XController*, which deals only with such events (there is no *adu* component).

Start, Abort and Suspend revisited

In the diagrammatic representation of the specification, start, suspend and abort gates were omitted since they did not exhibit any interesting behaviour in this example. All the interactors are composed synchronously on these gates, so in effect *start, suspend and abort* have assumed their roles for the composite interface. Connections between interactors were shown as directed arrows where there is a data flow. The arrow distinguishes the producer and the consumer of the data value. Signalling events are not directed in the same way, so producer and consumer components are not distinguished. In the diagram they are shown as simple lines connecting the components.

The suspend behaviour of used in this example was not correctly defined. An updated versions of the suspend-resume behaviour and a restart behaviour has been added to the standard dialogues. The following controller component, implements this structure and allows for constraints to define the interactor specific dialogue structure:

```
process controller [start, restart, suspend, resume, abort,...] : exit :=
      start; run [restart, suspend, resume, abort, inpPnt, ...]
endproc (* controller *)

process run[restart, suspend, resume, abort, ...] : exit :=
      (constraints[...]
```

---

2 It remains to be verified that this modification of the ADC model does not affect the compositionality property of the model.

```
      |[...]|
      susp_resum[restart, suspend, resume, abort,...] )
            [>
      interrupt[restart, suspend, resume, abort,...]
endproc

process constraints[...] : noexit :=
      ...
endproc

process interrupt[restart, suspend, resume, abort, ...]: exit :=
      restart; run[restart, suspend, resume, abort, ...]
      [] abort; exit
endproc

process susp_resum[restart, suspend, resume, abort, ...]
: noexit :=
      anyOrder[...]
      [>
      suspend; resume;  susp_resum[restart, suspend, resume, abort, ...]
endproc

process anyOrder[...] : noexit :=
      ...
endproc
```

Given the nature of the example, it is not quite clear how useful it is to equip each interactor with such standard parameterised dialogues. This needs to be investigated with further example specifications.

Abstraction-, Display- and Controller- only components

In the original model definition all interactors were expected to have an abstraction-display unit composed with a controller with the LOTOS synchronisation operator. In the specification discussed above, it became clear that special cases ought to be supported by the model. These can be thought of reduced or degenerate versions of the ADC model. In some cases, the interactor does not have a display status, it is called an abstraction-only interactor. Usually such interactors are not in direct interaction with the user. Alternatively it maintains no abstraction state, it is a display-only interactor, or it has no data handling capabilities at all and is a controller-only component .

For example, the monitor interactor is an abstraction-only interactor, the forward button is a display only interactor and XController is a controller only interactor. In the diagrammatic representation of the specification these families of interactor components are shown as different types of nodes on the graph.

Logical connectives

Interactors communicate with signals (pure synchronisation) and data. This communication is not always synchronous and one to one. Elsewhere [1] it was discussed how the multi-way synchronisation of LOTOS was useful as it enables the composition of constraints. In designing though the communication between interaction the syncrhonisation of LOTOS is restricting.

The specification of the receiving interactor ought to be independent of the interactors that send data, or just signals, to it. The synchrony of LOTOS might be overcome in practice by introducing logical connectives, such as the *demux* process used in the specification above. Since one of the aims of this exercise has been to investigate how interactors can be

composed to form a composite model of the interface software, a brief reflection as to what types of connections might be needed is useful.

In the implementation of user interfaces asynchronous communication is used more than synchronous communication. In some cases broadcasting mechanisms are used. A variety of event management techniques can be envisaged but this issue is more relevant in implementation architectures. Synchronous languages have better understood semantics, and are employed at higher level of abstractions, with the aim of analyisng a specification. Sometimes, it is useful to simulate asyncrhony or other event management schemes in the syncrhonous language. For the purposes of using the ADC model to construct user interfaces by composition, a few basic communication schemes need to be supported. Support in this context means that shorthands should be provided for their specifications in LOTOS, and that their use should be transparent to the specifier, and consistent with the manipulations of the specifications, and the composition graph representation of the specification.

Connections between ADC interactors represent flow of information. Communication over a particular gate, from a set of producers to a set of consumers mat be characterised by the involvement of all interactors in the set (AND) or one of them only (XOR). These logical relations between the interactors, help classify the basic communication schemes that are supported by the ADC model. Table 1 presents in LOTOS how the most interesting communication is specified in each case. The process *and1toN[g, g_1, ..., g_n]* should read a value at gate g and transmit it to gates $g_1..g_n$, ideally at any order. One possible specification is as follows:

```
process and1toN[g, g1, ..., gn]: noexit :=
      g?X:aSort;
      (send[g1](X) ||| ... ||| send[gn](X)) >> and1toN[g, g1, ..., gn]
endproc

process send[g](X:aSort) : noexit :=
      g!X; exit
endproc
```

In general the connective could itself be considered to be an interactor; sometimes a particular dialogue behaviour has to be supported. For example, in the specification presented, *playPauseACU* supports many to many communication between the interactors connected to it. The connectives discussed here arein fact templates of LOTOS code. In the composition graphs, they represent all the cases where communication may be represented simply by connecting lines between gates of the component interactors. It seems excessive to support other communication schemes not included in this set.

AND	**one consumer**	**many consumers**
one producer	P[g] |[g]| C[g]	P[g] |[g]| and1toN[g, g_1, ..., g_n] |[g, g_1, ...,g_n]| (C_1[g_1]|||C_2[g_2]|||...|||C_n[g_n])
many producers	(P_1[g]|[g]|...|[g] ||P_n[g]) |[g]| C[g]	(P_1[g]|[g]|.. |[g]|.||P_n[g]) |[g]| and1toN[g, g_1, ..., g_n] |[g, g_1, ...,g_n]| (C_1[g_1]|||C_2[g_2]|||...|||C_n[g_n])

XOR	**one consumer**	**many consumers**
one producer	P[g] |[g]| C[g]	P[g]|[g]|(C1[g]|||C2[g]|||...|||Cn[

many producers	$(P_1[g] \|\|\| P_2[g] \|\|\| ... \|\|\| P_n[g])$ $\|[g]\|$ $C[g]$	$(P_1[g] \|\|\| P_2[g] \|\|\| ... \|\|\| P_n[g])$ $\|[g]\|$ $(C_1[g] \|\|\| C_2[g] \|\|\| ... \|\|\| C_n[g])$

In the above tables it is implied that a [g] denotes a set G such that g∈ G. So when processes are combined in parallel g is implied that their synchronisation gate should at least include g. The process *and1toN* that is added in the AND composition, is an operationalisation in LOTOS of the logical requirement that all consumers should receive the same data value, and that the producer should proceed asynchronously with its operation. An serialised implementation of *and1toN* is *demux1* listed in an earlier section.

It is noted that in the case of the AND connection of many producers, they all synchronise. This might not be useful in practice. It would be a more useful construct, say, to simulate asynchrony. However, the aim of developing the ADC model is not to support a general purpose programming language. The requirements for the logical connectives is that they capture the necessary data flows, and that they are integrated into the transformations that are supported by the initial model (i.e. the compositions of ADC components). This is the subject of current development of the original ADC model, so that it encompasses the modifications introduced in this section.

With respect to their visual representations, the AND and XOR connections, when many consumers or producers are involved, are represented a special node. Different types of nodes (filled-in ellipse or an empty rectangle in the diagram of figure 4), may indicate whether the connection is an XOR or an AND connection.

11. Assessment of the study: lessons drawn and limitations

This report has given a full exposition of the reverse engineering case study, whose main aim, stated in the introduction of this report, has been to assess the ADC model. Indeed, many lessons were learnt and experience has been gained with a specification quite within the intended domain of the ADC model (graphical interaction), but also one that is quite challenging.

ADC has been shown to be an appropriate abstraction for interaction software. The specified interaction behaviour is in fact implemented on a radically different software architecture. The case study has demonstrated that it is feasible to specify interactive systems based as a composition of ADC interactors.

As a specification template, it was shown that all interactive components could be modelled as instantiations of the ADC interactor model. For the software engineer who wishes to formally specify an interface design, i.e. the potential user of this model, its existence reduces the specification effort by providing a ready conceptual structure, and a standard way of writing specifications.

While all interactor components could be written as instantiations of the ADC model, consistently doing so can be cumbersome. Special cases of ADC models were identified:

1. interactors without a display component
2. interactors without an abstraction component
3. interactors with only a control component

Strictly speaking an abstraction-only interactor is not consistent with the concept of the ADC interactor; after all the user cannot interact directly with them. All the special cases are interesting for practical purposes. A re-formulation of the formal definition to allow for these special cases was necessary.

The implicit requirements for modelling the communication between interactors, both in the ADC model [1] and its predecessor model [9] have not been addressed. For the purposes of modularity, and for the benefit of the specifier - user of the ADC model, it becomes necessary to introduce standard connectives between the ADC model. This is seen as a limitation of the initial formalisation model that was exposed by the study. The connectives themselves can be thought of as abstraction only interactors. They are necessary to support communication schemes not built in the composition operators of LOTOS.

It was seen in practice how the use of the interactors is greatly facilitated by the existence of an overall architectural model. One of the main difficulties in this study was to define what is the application interfaced to. Defining a behavioural specification of the functional core is a useful activity that defines the boundary for the use of the interactor model. It requires adopting an overall architecture like for example the IFIP reference model for interactive software [8].

Similarly the boundary of the scope of the interactive software needs to be defined at the display side. Serious and distracting repetition was observed in the specification of mouse level interactions. All interactors that receive input for the mouse included a dialogue constraint that is more naturally attributed to the mouse. In this case study they could have been omitted from all interactors, but then spurious behaviours would result e.g. successive mouse presses with no release.

Similarities were observed across components. Most components that the user interacts directly with included variations of the same constraints in their dialogue specification. These are consistent with the ADC model. To use in practice an interactor model like ADC, such common dialogue components have to be investigated, classified and provided as library components to the specifier. Some of the standard ones were: trigger specifications, input-output constraints, input only constraints. These were different for push buttons and dragging interactions, but within these groups they were quite standard.

Data type specifications can be superficially produced by syntactic substitution of sort names of the interactor instance with the general sort identifiers of the ADC model. However, such specifications lack in content. To relate more to the domain in question, and to provide meaningful specifications and simulations, case specific data types can be used. While in the case study, little re-use of such components was achieved (notably between pop up sliders, and between the buttons on the movie controller), it seems a promising idea to provide a library of abstract data types. This has been done in other fields of application of formal specifications, indeed LOTOS is packaged with a standard library for abstract data types e.g. integers, Booleans etc. The ADC model provides the architectural framework for identifying these components and for using them in the specification of user interfaces.

The level of abstraction of the specification was very low, modelling lexical level interactions. It was argued that this puts the specification technique to the test, as indeed was shown with the case of *XController*. If the model is used as the basis of using specification in design, the selection of re-usable components will greatly determine the level of abstraction of the final specification.

The case study has tested several aspects of the ADC model However, some parts of it remain untested as a result of the choice of specificand. The Simple Player™ application, provides an interaction rich in temporal dependencies, but the structure of the interactor itself is quite static. It involves the same set of interactive components throughout its

operation. As a counter example consider a drawing package where dynamic creation/destruction of interactive objects is required. Further, the interaction with Simple Player™ does not include navigation dialogue as is commonly found in hypertext systems, or form filling interfaces. These aspects may need to be tested with further small scale examples, of existing interfaces (reverse engineering) or at least some idealised design of them.

12. References

[1] Markopoulos P (1995) On the Expression of Interaction Properties within an Interactor Model. In Palanque P, Bastide R (eds), Design, Specification, Verification of Interactive Systems '95, Springer-Verlag.

[2] Apple Computer Inc. (1993). Inside Macintosh. QuickTime™. Addison Wesley.

[3] Bolognesi T & Brinksma E (1989) Introduction to the ISO specification language Lotos. In van Eijk P, Vissers C & Diaz M (eds) The Formal Description Technique Lotos, North-Holland, 1989.

[4] Vissers C A, Scollo G, van Sinderne M & Brinksma E (1991) Specification styles in distributed systems design and verification. Theoretical Computer Science 89, pp 179-206.

[5] Clark, R.G., LOTOS Design-Oriented Speicifications in the Object Based Style, Technical Report ,TR 84, , Dept. of Computer Science and Mathematics, University of Stirling, April 1992.

[6] Dix A J (1991) Formal Methods for Interactive Systems, Academic Press.

[7] Bolognesi T (1993) Deriving graphical representations of process networks from algebraic expressions. Information Processing Letters 46 (1993) 289-294.

[8] Bass, L., Cockton, G., Unger, C., IFIP Working Group 2.7. User Interface Engineering. A Reference Model for Interactive System Construction, 1993.

[9] Paterno F (1994) A Theory of User Interaction Objects. Journal of Visual Languages and Computing, 5, 227-249. Academic Press Ltd.

Annex 1.

Full Listing of the specification with the untimed model of the functional core.

```
(*  last change     23 Nov
   case study version 1.1 *)


specification interaction [video, data, init, actionPlay, actionPause,
actionGetMovieTime,
       out_ok, actionGotoTime, actionSetAllFrames,
       actionSetSelection, mcSetMovieBox, getMoviePicts, s, su, ab, click,
       doubleClick, playDSP, pauseDSP, display, iaDSP, iaV, d,  moveTHMB,
       press_shift, move_shift, release_shift, inpPLR, outPLR, iaPLRRct,
       iaPLRSg, iaPLRNt, inpDR, outDR, iaDR, oaDR,erase, enable, send,
       kbdSftPress, kbdSftRel, pressPLR, relPLR, dataPLR, pressFwd,relFwd,
pressBwd, relBwd, pressRate, relRate, pauseKbd, optionptionPlay, outTHMB, iaTHMB,
oaTHMB,pressBT, moveInBT,
       moveOutBT, releaseBT, outBT, iaBT, playBT, pauseBT,psPPCU, psXCNT,
       pressVOL, moveVOL, releaseVOL, outVOL, iaVOL,
       actionGetMovieVolume, actionSetVolume, playRT, moveInFwd,   moveOutFwd,
outFwd, gotoFwd, pauseBF, moveInBwd, moveOutBwdutBwd,  gotoBwd,
gotoBeginningOfMovie, gotoEndOfMovie, dataMN,  iaGc, pressBox, moveBox,relBox,
outBox, moveRate, outRate]: noexit


library Boolean, Integer  endlib

type segmentType is Integer
sorts segment
opns
       mkSegment            :      Int, Int      ->     segment
       start, duration             :      segment       ->     Int
       setStart, setDuration       segment, Int ->
segment
eqns
forall s:segment, m,n:int
ofsort int
       start(setStart(s,n))=n;
       start(setDuration(s,n))=start(s);
       duration(setStart(s,n))=duration(s);
       duration(setDuration(s,n))=n;
ofsort segment
       setStart(setStart(s,m),n)=setStart(s,n);
       setDuration(setDuration(s,m),n) = setDuration(s,n);
       setStart(setDuration(s,m),n)= setDuration(setStart(s,n),m);
endtype

type BoundedVal is Integer
sorts
       boundValue
opns
       default      :                              ->     boundValue
       lo, val, hi  :      boundValue             ->     Int
       setVal       :      boundValue, Int               ->     boundValue
       incr, decr   :      boundValue             ->     boundValue
eqns
forall b:boundValue, n: Int
ofsort Int
       lo(default) = 0;
       hi(default) = s(0);
       val(default) = 0;
```

```
      val(setVal(b, n)) = n;
      val(incr(setVal(b,n))) = s(n);
      lo(setVal(b,n)) = lo(b);
      hi(setVal(b,n))= hi(b);
ofsort bool
      lo(b) le val(b) = true;
      val(b) le hi(b) = true;
endtype

type graphics is Boolean
sorts pnt, rct, line, area, icon, hilite
opns
      isIn:  pnt, rct      ->     Bool
      mkLne: pnt, pnt      ->     line
      offset: area, pnt       ->      area
      drag:   rct, pnt     ->     rct
eqns
forall r1,r2:rct, p1,p2:pnt, a1,a2:area
ofsort rct
      drag(drag(r1,p1),p2)=drag(r1,p2);
ofsort area
      offset(offset(a1,p1),p2)=offset(a1,p2);
endtype


type movieData is segmentType, graphics
sorts movie, frame, still, sound, MData
opns
(*Movie(movieTime, duration, volume, rate, selection, activeSegment,movieData)*)
      movieTime, duration, volume, rate:movie        ->     Int
      selection, activeSegment:  movie         ->     segment
      allFrames            :     movie         ->     Bool
      movieBox             :     movie         ->     rct

      setMovieTime, setDuration: movie, Int    ->     movie
      setVolume, setRate   :     movie, Int    ->     movie
      setSelection         :     movie, segment->     movie
      setActiveSegment     :     movie, segment->     movie
      setMovieBox          :     movie, rct    ->     movie
(* some convenience actions *)
      incr                 :     movie         ->     movie
      decr                 :     movie         ->     movie
(* syntactic only definition of still and video frames *)
      getMoviePict         :     movie         ->     still
      videoFrame           :     movie         ->     frame
      setAllFrames         :     movie, Bool   ->     movie
eqns
forAll M: movie, b: bool, x,y:Int, s:segment, r:rct
ofsort movie
      incr(decr(M)) = M;
      decr(incr(M)) = M;
      setMovieTime(setMovieTime(M,x),y)=setMovieTime(M,y);
ofsort Int
      movieTime(setMovieTime(M,x))=x;
      movieTime(setDuration(M,x)) = movieTime(M);
      movieTime(setVolume (M, x)) = movieTime (M);
      movieTime(setRate(M, x)) = movieTime (M);
      movieTime(setActiveSegment(M, s)) = start(s);
      movieTime(setSelection(M, s)) = movieTime(M);
      movieTime(incr(M)) = movieTime(M) + s(0);
      movieTime(decr(M)) = movieTime(M) - s(0);
      movieTime(setAllFrames(M,b))=movieTime(M);
      movieTime(setMovieBox(M,r))=movieTime(M);

      duration(setMovieTime(M, x)) = duration(M);
      duration(setDuration(M, x)) = x;
      duration(setRate(M,x))=duration(M);
      duration(setVolume(M,x))= duration(M);
      duration(setActiveSegment(M, s)) = duration(s);
```

```
        duration(setSelection(M,s))=duration(M);
        duration(incr(M)) = duration(M);
        duration(decr(M)) = duration(M);
        duration(setAllFrames(M,b))=duration(M);
        duration(setMovieBox(M,r))=duration(M);

        volume(setMovieTime(M,x)) = volume(M);
        volume(setDuration(M, x))=volume(M);
        volume(setRate(M,x))=volume(M);
        volume(setVolume(M,x))= x;
        volume(setActiveSegment(M,s))=volume(M);
        volume(setSelection(M, s))= volume(M);
        volume(incr(M))=volume(M);
        volume(decr(M))=volume(M);
        volume(setAllFrames(M,b))=volume(M);
        volume(setMovieBox(M,r))=volume(M);

        rate(setMovieTime(M,x)) = rate(M);
        rate(setDuration(M, x))=rate(M);
        rate(setRate(M,x))=x;
        rate(setVolume(M,x))= rate(M);
        rate(setActiveSegment(M,s))=rate(M);
        rate(setSelection(M, s))= rate(M);
        rate(incr(M))= rate(M);
        rate(decr(M))=rate(M);
        rate(setAllFrames(M,b))=rate(M);
        rate(setMovieBox(M,r))=rate(M);
ofsort Bool
        allFrames(setMovieTime(M,x))=allFrames(M);
        allFrames(setDuration(M,x))=allFrames(M);
        allFrames(setRate(M,x))=allFrames(M);
        allFrames(setVolume(M,x))=allFrames(M);
        allFrames(setActiveSegment(M,s))=allFrames(M);
        allFrames(setSelection(M,s))=allFrames(M);
        allFrames(incr(M))=allFrames(M);
        allFrames(decr(M))=allFrames(M);
        allFrames(setAllFrames(M, b)) = b;
        allFrames(setMovieBox(M,r))=allFrames(M);
ofsort segment
        activeSegment(setMovieTime(M,x))=activeSegment(M);
        activeSegment(setDuration(M,x))=activeSegment(M);
        activeSegment(setRate(M,x))=activeSegment(M);
        activeSegment(setVolume(M,x))=activeSegment(M);
        activeSegment(setActiveSegment(M,s))=s;
        activeSegment(setSelection(M,s))=activeSegment(M);
        activeSegment(incr(M))=activeSegment(M);
        activeSegment(decr(M))=activeSegment(M);
        activeSegment(setAllFrames(M, b)) = activeSegment(M);
        activeSegment(setMovieBox(M,r))=activeSegment(M);

        selection(setDuration(M,x))=selection(M);
        selection(setRate(M,x))=selection(M);
        selection(setVolume(M,x))=selection(M);
        selection(setActiveSegment(M,s))=selection(M);
        selection(setSelection(M,s))=s;
        selection(incr(M))=selection(M);
        selection(decr(M))=selection(M);
        selection(setAllFrames(M, b)) = selection(M);
        selection(setMovieBox(M,r))=selection(M);
ofsort rct
        movieBox(setMovieTime(M, x)) = movieBox(M);
        movieBox(setDuration(M, x)) = movieBox(M);
        movieBox(setRate(M,x))=movieBox(M);
        movieBox(setVolume(M,x))= movieBox(M);
        movieBox(setActiveSegment(M, s)) = movieBox(M);
        movieBox(setSelection(M,s))=movieBox(M);
        movieBox(incr(M)) = movieBox(M);
        movieBox(decr(M)) = movieBox(M);
        movieBox(setAllFrames(M,b))=movieBox(M);
```

```
      movieBox(setMovieBox(M,r))=r;
endtype


type dsp_d is movieData, graphics
sorts
      disp
opns
      renderR:      disp, rct          ->     disp
      renderF:      disp, frame        ->     disp
      renderS:      disp, still        ->     disp
endtype


type playBarType is graphics, segmentType
sorts
      playBar
opns
      changePnt:    playBar, pnt       ->     playBar
      changeRect:   playBar, rct       ->     playBar
      changeLne:    playBar, line      ->     playBar
      changeScale:  playBar, Int       ->     playBar
      rect   :      playBar            ->     rct
      point  :      playBar            ->     pnt
      line   :      playBar            ->     line
      scale  :      playBar            ->     Int

      pntToInt:     playBar, pnt       ->     Int
      intToPnt:     playBar, Int       ->     pnt
      segToLne:     playBar, segment   ->     line
      lneToSeg:     playBar, line      ->     segment
eqns
      forall p,q:pnt, r,s:rct, ln, lm:line, pb:playBar, n,m:Int
ofsort rct
      rect(changeRect(pb,r)) = r;
      rect(changeLne(pb,ln)) = rect(pb);
      rect(changePnt(pb,p)) = rect(pb);
ofsort pnt
      point(changeRect(pb,r)) = point(pb);
      point(changeLne(pb,ln))= point(pb);
      point(changePnt(pb,p)) = p;
ofsort line
      line(changeRect(pb,r))= line(pb);
      line(changeLne(pb,ln))= ln;
      line(changePnt(pb,p))= line(pb);
ofsort Int
      scale(changeScale(pb, n))= n;
      scale(changeLne(pb, ln))=scale(pb);
ofsort playBar
      changeRect(changeRect(pb,r),s) = changeRect(pb,s);
      changeLne(changeLne(pb,ln), lm) = changeLne(pb,lm);
      changePnt(changePnt(pb,p), q) = changePnt(pb,q);
      changeScale(changeScale(pb,n),m)=changeScale(pb,m);
      changeRect(changeScale(pb,n),r)=changeScale(changeRect(pb,r),n);
      changeRect(changeLne(pb,ln),r)=changeLne(changeRect(pb,r),ln);
      changeRect(changePnt(pb,p),r)=changePnt(changerect(pb,r),p);
      changeScale(changeLne(pb,ln),n)=changeLne(changeScale(pb,n),ln);
      changeScale(changePnt(pb,p),n)=changePnt(changeScale(pb,n),p);
      changeLne(changePnt(pb,p),ln)=changePnt(changeLne(pb,ln),p);
(* scale has been defined syntactically only:  it changes implicitly
with changing the Rectangle, or by setting it explictily with an
outside command.  When the scale changes, both interpretation
functions, pntToInt, and intToPnt change with it.  This is again not
specified here. *)
endtype (* playerBar *)

type plr_ad is playBarType
sorts playBarData
opns
```

Formal Specification of the Simple Player ™ graphical interface using the ADC interactor model .

```
inputM:       pnt, playBar, playBarData  ->     playBarData
       inputMK: line, playBar, playBarData->     playBarData

       echoM: pnt, playBar, playBarData  ->     playBar
       echoMK:line, playBar, playBarData ->     playBar

       renderS:playBar, segment   ->     playBar
       renderT:playBar, Int       ->     playBar
       renderRct:playBar, rct             ->     playBar

       receiveS: playBarData, segment    ->     playBarData
       receiveT: playBarData, Int ->     playBarData
       receiveRct: playBarData, rct->     playBarData

       resultT: playBarData       ->     Int
       resultS: playBarData       ->     segment
eqns
forall r:rct, p:pnt, ln:line, pb: playBar, ts:playBarData, s:segment, t:Int
ofsort Int
       resultT(inputM(p,pb,ts)) = pntToInt(pb,p);
       resultT(inputMK(ln,pb,ts))=resultT(ts);
       resultT(receiveS(ts,s))=resultT(ts);
       resultT(receiveT(ts,t))=t;
       resultT(receiveRct(ts,r))=resultT(ts);
ofSort segment
       resultS(inputM(p,pb,ts)) = resultS(ts);
       resultS(inputMK(ln,pb,ts))= lneToSeg(pb, ln);
       resultS(receiveS(ts,s))=s;
       resultS(receiveT(ts,t))=resultS(ts);
       resultS(receiveRct(ts,r))=resultS(ts);
ofSort playBar
       echoM(p, pb, ts) = changePnt(pb, p);
       echoMK(ln,pb,ts) = changeLne(pb, ln);
       renderS(pb,s) = changeLne(pb, segToLne(pb,s));
       renderT(pb,t) = changePnt(pb, intToPnt(pb, t));
       renderRct(pb,r) = changeRect(pb,r);
endtype


type selection_abs is graphics, Boolean
opns
       nil:                              ->     line
       reset:              line          ->     line
       setStart, setEnd:   line, pnt     ->     line

       getPoint:           line, pnt     ->     line
       startSel, endSel:   line          ->     pnt

       isNil:              line          ->     Bool
       _ leftOf _:         pnt, line     ->     Bool
       _ rightOf _:        pnt, line     ->     Bool
eqns
forall d,d1,d2:line, p,p1,p2:pnt
ofsort Bool
       isNil(nil)=true;
       isNil(setStart(d,p))=false;
       isNil(setEnd(d,p))=false;
       isNil(getPoint(d,p))=false;
ofsort pnt
       startSel(setStart(d,p))=p;
       endSel(setStart(d,p))=endSel(d);
       startSel(setEnd(d,p))=startSel(d);
       endSel(setEnd(d,p))=p;
ofsort line
       setStart(setStart(d,p1),p2)=setStart(d,p2);
       setEnd(setEnd(d,p1),p2)=setEnd(d,p2);
       setStart(setEnd(d,p1),p2)=setEnd(setStart(d,p2),p1);
       reset(d)=nil;
       getPoint(nil, p) = setEnd(setStart(nil, p),p);
```

```
      not(isNil(d)) and (startSel(d) leftOf d) =>
            getPoint(d,p) = setStart(d,p);
      not(isNil(d)) and (startSel(d) rightOf d) =>
            getPoint(d,p) = setEnd(d,p);
endtype

type selection_dsp is playBarType
opns
      hilite:       playBar, line ->     playBar
      dehilite:     playBar       ->     playBar
eqns
forall pb:playBar, l,m:line
ofsort playBar
      hilite(hilite(pb,l),m)=hilite(pb,m);
      hilite(dehilite(pb),m)=hilite(pb,m);
      dehilite(hilite(pb,m))=pb;
endtype

type selection_ad is selection_abs, selection_dsp
opns
      inputPnt:     pnt, line            ->     line
      inputEr:      line                 ->     line
      echo:         pnt, playBar, line   ->     playBar
      echoEr:       playBar, line        ->     playBar
      receivePB:    line, playBar        ->     line
      renderPB:     playBar, playBar     ->     playBar
eqns
forall p:pnt, a:line, pb,pb1,pb2:playBar
ofsort  line
      inputPnt(p,a) = getPoint(a,p);
      inputEr(a)=nil;
      receivePB(receivePB(a,pb1),pb2)=receivePB(a,pb2);
ofsort playBar
      renderPB(pb1,pb2) = pb2;
      echo(p,pb,a)=hilite(pb,getPoint(a,p));
      echoEr(pb,a)=dehilite(pb);
endtype

type thumbDisplayType is playBarType
sorts
      thumb_dsp
opns
      moveThumb:    thumb_dsp, pnt       ->     thumb_dsp
      2Dto1Dpnt:    thumb_dsp, pnt       ->     pnt
eqns
forall td: thumb_dsp, pnt1, pnt2: pnt
ofsort thumb_dsp
      moveThumb(moveThumb(td,pnt1),pnt2)=moveThumb(td, pnt2);
(* the display has no memory of previous state - there is only one
thumb displayed on the screen *)
endtype

type thumb_ad is Graphics, playBarType, thumbDisplayType
opns
      inputPress    :      pnt, thumb_dsp, pnt ->     pnt
      inputMove     :      pnt, thumb_dsp, pnt ->     pnt
      inputRelease  :      pnt, thumb_dsp, pnt ->     pnt
      echoPress     :      pnt, thumb_dsp, pnt ->     thumb_dsp
      echoMove      :      pnt, thumb_dsp, pnt ->     thumb_dsp
      echoRelease   :      pnt, thumb_dsp, pnt ->     thumb_dsp
        render      :      thumb_dsp, playBar      -> thumb_dsp
        receive             :      pnt, playBar           -> pnt
        result      :      pnt                  ->     pnt
eqns
forall a,p:pnt, d:thumb_dsp, pb:playBar
ofsort  pnt
        inputPress(p, d, a) =  2Dto1Dpnt(d,p);
        inputMove(p, d, a) = 2Dto1Dpnt(d,p);
        inputRelease(p, d, a) = 2Dto1Dpnt(d,p);
```

Formal Specification of the Simple Player ™ graphical interface using the ADC interactor model .

```
ofsort thumb_dsp
        echoPress(p, d, a) = moveThumb(d,p);
        echoRelease(p, d, a) = moveThumb(d,p);
        echoMove(p, d, a) = moveThumb(d,p);
        render(d, pb) = moveThumb(d,point(pb));
ofsort pnt
       receive(a, pb) = point(pb);
        result(a) = a;
endtype

type twoStateButton is graphics
sorts
       twoStateButton_dsp
opns
       mkButtonDsp: icon, rct, hilite   ->     twoStateButton_dsp
       getIcon:     twoStateButton_dsp          ->     icon
       getHilite:   twoStateButton_dsp          ->     hilite
       getRect:     twoStateButton_dsp          ->     rct
       playIcon, pauseIcon:                     ->     icon
       onH, offH:                               ->     hilite
       setIcon:     twoStateButton_dsp, icon    ->     twoStateButton_dsp
       setRect:     twoStateButton_dsp, rct     ->     twoStateButton_dsp
       setHilite:   twoStateButton_dsp, hilite  ->     twoStateButton_dsp
       switch:      twoStateButton_dsp          ->     twoStateButton_dsp
eqns
forall ppd:twoStateButton_dsp, r,s:rct, cn, cm:icon, h,g:hilite
ofsort rct
       getRect(mkButtonDsp(cn, r, h))=r;
       getRect(setRect(ppd, r))=r;
       getRect(setHilite(ppd, h))=getRect(ppd);
       getRect(setIcon(ppd, cn))=getRect(ppd);
       getRect(switch(ppd)) = getRect(ppd);
ofsort icon
       getIcon(mkButtonDsp(cn, r, h))=cn;
       getIcon(setRect(ppd, r))=getIcon(ppd);
       getIcon(setHilite(ppd, h))=getIcon(ppd);
       getIcon(setIcon(ppd, cn))=cn;
       getIcon(ppd) = playIcon => getIcon(switch(ppd)) = pauseIcon;
       getIcon(ppd) = pauseIcon => getIcon(switch(ppd)) = playIcon;
ofsort hilite
       getHilite(mkButtonDsp(cn, r, h))=h;
       getHilite(setRect(ppd, r))=getHilite(ppd);
       getHilite(setHilite(ppd, h)) = h;
       getHilite(setIcon(ppd, cn))=getHilite(ppd);
       getHilite(switch(ppd))= getHilite(ppd);
ofsort twoStateButton_dsp
       switch(switch(ppd))=ppd;
       setHilite(switch(ppd),h)   = switch(setHilite(ppd, h));
       setIcon(switch(ppd),cn)    = setIcon(ppd, cn);
       setRect(switch(ppd),r)           = switch(setRect(ppd, r));
       setHilite(setIcon(ppd,cn),h)= setIcon(setHilite(ppd,h),cn);
       setHilite(setRect(ppd,r),h)= setRect(setHilite(ppd,h),r);
       setRect(setIcon(ppd,cn),r) = setIcon(setRect(ppd,r),cn);
       setHilite(setHilite(ppd,h),g)     = setHilite(ppd,g);
       setIcon(setIcon(ppd,cn),cm)= setIcon(ppd,cm);
       setRect(setRect(ppd,r),s)  = setRect(ppd,s);
endtype

type playPause_d is twoStateButton
opns
       echoPr :     twoStateButton_dsp          ->     twoStateButton_dsp
       echoRel:     twoStateButton_dsp          ->     twoStateButton_dsp
       echoMovIn:   twoStateButton_dsp          ->     twoStateButton_dsp
       echoMovOut:  twoStateButton_dsp          ->     twoStateButton_dsp
         renderR    :     twoStateButton_dsp, rct     ->     twoStateButton_dsp
       renderS:     twoStateButton_dsp          ->     twoStateButton_dsp
eqns
forall p:pnt, ppd:twoStateButton_dsp, r:rct
ofsort twoStateButton_dsp
```

```
        echoPr(ppd)   = setHilite(ppd, onH);
        echoMovIn(ppd) = setHilite(ppd, onH);
        echoMovOut(ppd) = setHilite(ppd, offH); ;
        echoRel(ppd) = setHilite(switch(ppd),offH);
        renderR(ppd,r)       = setRect(ppd,r);
        renderS(ppd)  = switch(ppd);
endtype

type popUpSlider is graphics, Integer
sorts
        volumeBar
opns
        popUpSlider:          volumeBar     ->     volumeBar
        chIconAndSlider:      volumeBar, pnt->     volumeBar
        popDownSlider:        volumeBar     ->     volumeBar
        changeRect:           volumeBar, rct->     volumeBar
        pntToInt:             volumeBar, pnt->     Int
        IntToBar:             volumeBar, Int->     volumeBar
eqns
forall vb:volumeBar, p,q:pnt, r,s:rct, v,w:Int
ofsort volumeBar
        popDownSlider(popUpSlider(vb)) = popDownSlider(vb);
        popDownSlider(popDownSlider(vb))=popDownSlider(vb);
        popUpSlider(popUpSlider(vb))=popUpSlider(vb);
        popUpSlider(popDownSlider(vb))=popUpSlider(vb);
        chIconAndSlider(chIconAndSlider(vb,p),q)=chIconAndSlider(vb,q);
        chIconAndSlider(popUpSlider(vb),p)=popUpSlider(chIconAndSlider(vb,p));
        chIconAndSlider(popDownSlider(vb),p)=popDownSlider(chIconAndSlider(vb,p));
        changeRect(changeRect(vb,r),s)=changeRect(vb,s);
        changeRect(popDownSlider(vb),r)=popDownSlider(changeRect(vb,r));
        changeRect(popUpSlider(vb),r) = popUpSlider(changeRect(vb,r));
        changeRect(chIconAndSlider(vb,p),r)=chIconAndSlider(changeRect(vb,r),p);
        IntToBar(popUpSlider(vb),v)=popUpSlider(IntToBar(vb,v));
        IntToBar(popDownSlider(vb),v)=popDownSlider(IntToBar(vb,v));
        IntToBar(IntToBar(vb,v),w)=IntToBar(vb,w);
        IntToBar(chIconAndSlider(vb, p),v)=IntToBar(vb,v);
        IntToBar(changeRect(vb, r), v)= changeRect(IntToBar(vb,v),r);
ofsort Int
        pntToInt(popUpSlider(vb), p)=pntToInt(vb,p);
        pntToInt(popDownSlider(vb),p)=pntToInt(vb,p);
        pntToInt(chIconAndSlider(vb,q),p)=pntToInt(vb,p);
endtype

type pushButtonType is twoStateButton renamedby
sortnames pb_dsp for twoStateButton_dsp
endtype

type resizeButton_ad is pushButtonType
opns
        echoPr :      pb_dsp, rct, pnt     ->     pb_dsp
        echoRel:      pb_dsp, rct, pnt     ->     pb_dsp
        echoMove:     pb_dsp, rct, pnt     ->     pb_dsp
        inpPr  :      pb_dsp, rct, pnt     ->     rct
        inpRel :      pb_dsp, rct, pnt     ->     rct
        inpMov :      pb_dsp, rct, pnt     ->     rct
        resultRB  :   rct                  ->     rct
eqns
forall p:pnt, ppd:pb_dsp, r:rct
ofsort pb_dsp
        echoPr(ppd,r, p)     = setHilite(ppd, onH);
        echoMove(ppd,r,p)    = setRect(ppd, drag(r,p));
        echoRel(ppd,r,p)     = setHilite(ppd, offH);
ofsort rct
        inpPr(ppd, r, p)=r;
        inpMov(ppd,r,p)= drag(r,p);
        inpRel(ppd, r, p)=drag(r,p);
        resultRB(r)=r;
endtype
```

```
type volume_ad is popUpSlider
opns
       inputPr:      volumeBar, Int       ->     Int
       echoPr:       volumeBar, Int       ->     volumeBar
       inputMov:     pnt, volumeBar,Int   ->     Int
       echoMov:      pnt, volumeBar, Int  ->     volumeBar
       inputRel:     volumeBar,Int        ->     Int
       echoRel:      volumeBar, Int       ->     volumeBar
       renderRV:     volumeBar, rct       ->     volumeBar
       receiveRV:    Int, rct             ->     Int
       renderV:      volumeBar, Int       ->     volumeBar
       receiveV:     Int, Int             ->     Int
       result:       Int                  ->     Int

eqns
forall r:rct, p:pnt, v, n:Int, vb:volumeBar
ofsort Int
       result(v) = v;
       inputPr(vb,v) = v;
       inputMov(p,vb,v) = pntToInt(vb,p);
       inputRel(vb,v) = v;
       receiveRV(v,r)=v;
       receiveV(v,n)=n;
ofSort volumeBar
       echoPr(vb, v) = popUpSlider(vb);
       echoMov(p, vb, v) = chIconAndSlider(vb,p);
       echoRel(vb, v) = popDownSlider(vb);
       renderRV(vb,r) = changeRect(vb,r);
       renderV(vb,v) = IntToBar(vb,v);
endtype

type rate_ad is volume_ad renamedby
sortnames
       rateBar for volumeBar
opnnames
       renderRt for renderRV
       receiveRt for receiveRV
       renderR for renderV
       receiveR for receiveV
endtype

type timeInfo is Integer
sorts
       montr_abs
opns
       now:   montr_abs            ->     int
       dt:    montr_abs            ->     int
       end:   montr_abs            ->     int
       setNow:montr_abs, int       ->     montr_abs
       setDt: montr_abs, int       ->     montr_abs
       setEnd:montr_abs, int       ->     montr_abs
eqns
forall m,n:int, mab:montr_abs
ofsort int
       now(setNow(mab,n))=n;
       now(setDt(mab,n))=now(mab);
       now(setEnd(mab,n))=now(mab);
       dt(setDt(mab,n))=n;
       dt(setNow(mab,n))=dt(mab);
       dt(setEnd(mab,n))=dt(mab);
       end(setEnd(mab,n))=n;
       end(setNow(mab,n))=end(mab);
       end(setDt(mab,n))=end(mab);
ofsort montr_abs
       setNow(setNow(mab,m),n)=setNow(mab,n);
       setEnd(setEnd(mab,m),n)=setEnd(mab,n);
       setDt(setDt(mab,m),n)=setDt(mab,n);
       setNow(setDt(mab,n),m)= setDt(setNow(mab,m),n);
```

```
        setEnd(setDt(mab,n),m) = setDt(setEnd(mab,m),n);
        setEnd(setNow(mab,m),n)=setNow(setEnd(mab,n),m);
endtype

type montr_ad is timeInfo
opns
        receive:        montr_abs, int      ->      montr_abs
        receiveD:       montr_abs, int      ->      montr_abs
        jumpF:          montr_abs           ->      montr_abs
        jumpB:          montr_abs           ->      montr_abs
        inputF:         montr_abs           ->      montr_abs
        inputB:         montr_abs           ->      montr_abs
        result:         montr_abs           ->      int
eqns forall t:Int, m:montr_abs
ofsort Int
        result(m) = now(m)+dt(m);
ofsort montr_abs
        receiveD(m,t) = setEnd(m,t);
        receive(m,t)= setNow(m,t);
        jumpF(m) = setNow(setDt(m,0),end(m));
        jumpB(m) =  setNow(setDt(m,0),0);
(now(m) lt end(m))=>
        inputF(m) = setDt(m, s(0));
(0 lt now(m))=>
        inputB(m) = setDt(m, p(0));
 (now(m) eq end(m))
        inputF(m) = m;
endtype


type defaults is movieData, selection_ad, thumb_ad, playPause_d,
                volume_ad, montr_ad, rate_ad, resizeButton_ad, plr_ad, dsp_d
opns

        noSelection:            ->      line
        initPlayBarData:        ->      playBarData
        thePlayerBar:           ->      playBar
        indicator:              ->      pnt
        aTime:                  ->      montr_abs
        thumb  :                ->      thumb_dsp
        play_icon:              ->      twoStateButton_dsp
        fwd_arrows_icon:        ->      twoStateButton_dsp
        bwd_arrows_icon:        ->      twoStateButton_dsp
        defaultVolumeBar:       ->      volumeBar
        preferredVolume:        ->      Int
        resizeBox:              ->      pb_dsp
        graphicalContext:       ->      rct
        defaultRate:            ->      int
        defaultRateBar:                 ->      rateBar
        movieDisplay:           ->      disp
eqns
ofsort int
        dt(aTime) = 0;
        now(aTime) = 0;
        end(aTime) = s(s(s(0)));
endtype


behaviour

fnctCore[actionPlay, actionPause,  actionGetMovieTime, out_ok, actionGotoTime,
gotoBeginningOfMovie, gotoEndOfMovie, actionSetAllFrames,
                actionSetSelection, actionSetVolume, actionGetMovieVolume, mcSetMovieBox,
video, data, init]

|[actionPlay, actionPause,  actionGetMovieTime, out_ok, actionGotoTime,
gotoBeginningOfMovie, gotoEndOfMovie, actionSetAllFrames,
actionSetSelection, actionSetVolume, actionGetMovieVolume, mcSetMovieBox, video, data]|
```

Formal Specification of the Simple Player ™ graphical interface using the ADC interactor model .

```
(demux4[data, dataPLR, dataMN]
|[dataPLR, dataMN]|
(((  ( (      playPauseACU[s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP, psPPCU, pauseKbd,
                                                pauseBF, iaDSP, iaBT, optionPlay,
actionSetAllFrames]
          |[s, su, ab, playDSP, pauseDSP, playBT, psPPCU, iaBT, iaDSP]|
              (      dsp[s, su, ab, click, doubleClick, display, iaDSP, mcSetMovieBox,
video, out_ok, playDSP, pauseDSP]
             |[s, su, ab, mcSetMovieBox]|
                    (      playPauseButton[s, su, ab, pressBT, moveInBT,
moveOutBT,releaseBT,
                                  outBT, mcSetMovieBox, iaBT, playBT, pauseBT]
                    |[pauseBT]|  demux2[pauseBT, psPPCU, psXCNT])))
       |[s, su, ab, mcSetMovieBox]|

       (      (      (      plr[s, su, ab, inpPLR, oaDR, outPLR, mcSetMovieBox,
dataPLR,
                                                dataPLR, actionGotoTime,
actionSetSelection]
                           |[outPLR]| demux1[outPLR, iaTHMB, iaDR])
                    |[s, su, ab, iaTHMB, inpPLR]|
                    (      thumb[s, su, ab, pressPLR, moveTHMB, relPLR, outTHMB,
iaTHMB, oaTHMB]

                           |[oaTHMB]| demux3[oaTHMB, inpPLR, inpDR]))
             |[s, su, ab, inpDR, iaDR, oaDR]|
                    selectionBand[s, su, ab, inpDR, outDR, iaDR, oaDR, erase, enable,
send])

     |[s, su, ab, pauseBF, mcSetMovieBox]|


       (volume[s, su, ab, pressVOL, moveVOL, releaseVOL, outVOL, mcSetMovieBox,
actionGetMovieVolume, actionSetVolume]
       |[s, su, ab, mcSetMovieBox]|

             (      monitor[s, su, ab, option, actionGetMovieTime, actionGotoTime,
gotoFwd,
                                          gotoEndOfMovie, gotoBwd,
gotoBeginningOfMovie, dataMN]
             |[s, su, ab, gotoFwd, gotoBwd]|
                    (      pushButton[s, su, ab, pressFwd, moveInFwd, moveOutFwd,
relFwd,
                                  outFwd, mcSetMovieBox, gotoFwd,
pauseBF](fwd_arrows_icon)
                    |[s, su, ab, mcSetMovieBox]|
                           pushButton[s, su, ab, pressBwd, moveInBwd, moveOutBwd,
relBwd,
                                  outBwd, mcSetMovieBox, gotoBwd,
pauseBF](bwd_arrows_icon)))))
    |[s, su, ab, mcSetMovieBox, pressFwd, relFwd, pressBwd, relBwd,
      pressBT, psXCNT, pauseKbd, erase, send, enable, pressPLR, relPLR]|
       xcontroller[s, su, ab, kbdSftPress, kbdSftRel, pressBT, psXCNT, pressPLR, relPLR,
                   pressFwd, relFwd, pressBwd, relBwd, pressRate, relRate, pauseKbd,
                   enable, send, erase])
 |[s, su, ab, mcSetMovieBox, playRT, pressRate, relRate]|
      rate[s, su, ab, pressRate, moveRate, relRate, outRate, mcSetMovieBox, playRT] )
|[s, su, ab, mcSetMovieBox]|
resize[s, su, ab, pressBox, moveBox, relBox, outBox, mcSetMovieBox])
                                  )


where


process demux1[outPLR, iaTHMB, iaDR]: noexit :=
      outPLR?x:playBar;
```

```
        (       iaTHMB?y:playBar[x=y];      iaDR?z:playBar[z=x];demux1[outPLR, iaTHMB,
iaDR]
        []      iaDR?z:playBar[z=x];iaTHMB?y:playBar[x=y];      demux1[outPLR, iaTHMB,
iaDR])
endproc (* a logical AND behaviour *)

process demux2[pauseBT, psPPCU, psXCNT]: noexit :=
        pauseBT;        (psPPCU; psXCNT; demux2[pauseBT, psPPCU, psXCNT]
                        [] psXCNT; psPPCU; demux2[pauseBT, psPPCU, psXCNT])
endproc (* an AND behaviour *)

process demux3[oaTHMB,thmbPos1, thmbPos2]: noexit :=
        oaTHMB?x:pnt;
        (       thmbPos1?y:pnt[x=y]; thmbPos2?z:pnt[z=x]; demux3[oaTHMB, thmbPos1,
thmbPos2]
        []      thmbPos2?z:pnt[z=x]; thmbPos1?y:pnt[x=y]; demux3[oaTHMB, thmbPos1,
thmbPos2])
endproc (* an AND behaviour *)

process demux4[d,d1,d2]: noexit :=
        d?x:int;
                (       d1?y:int[x=y]; d2?z:int[z=x]; demux4[d,d1,d2]
                []      d2?z:int[z=x]; d1?y:int[x=y]; demux4[d,d1,d2])
        []
        d?x:segment; d1?y:segment [x=y]; demux4[d,d1,d2]
endproc (* an AND behaviour *)

(*******************playPauseACU *******************************)
process playPauseACU[s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]: noexit
:=
        au[playRT, actionPlay](defaultRate)
        |[playRT, actionPlay]|
        controller[s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
where (* acu *)

process controller [s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll] : exit :=

s; run [su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]

where (* controller *)

process run[su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]: exit :=
        constraints[playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
                [>
        suspend [su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
where  (* run *)

process constraints[playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]:noexit :=

(   (playDSP; exit [] playBT; exit []  playRT?x:int; exit [] option; playBT; setAll;
exit) >>
        (actionPlay?y:int;    (iaBT; iaDSP; exit [] iaDSP; iaBT; exit) >>
                (       changesOfRate[playRT, actionPlay]
                [>
                        (choice B in [pauseDSP, pauseBT, pauseKbd, pauseBF]
                                [] B; actionPause; (iaBT; iaDSP; exit [] iaDSP; iaBT;
exit))))
 []
```

```
        (       iaTHMB?y:playBar[x=y];      iaDR?z:playBar[z=x];demux1[outPLR, iaTHMB,
iaDR]
        []      iaDR?z:playBar[z=x];iaTHMB?y:playBar[x=y];      demux1[outPLR, iaTHMB,
iaDR])
endproc (* a logical AND behaviour *)

process demux2[pauseBT, psPPCU, psXCNT]: noexit :=
        pauseBT;        (psPPCU; psXCNT; demux2[pauseBT, psPPCU, psXCNT]
                        [] psXCNT; psPPCU; demux2[pauseBT, psPPCU, psXCNT])
endproc (* an AND behaviour *)

process demux3[oaTHMB,thmbPos1, thmbPos2]: noexit :=
        oaTHMB?x:pnt;
        (       thmbPos1?y:pnt[x=y]; thmbPos2?z:pnt[z=x]; demux3[oaTHMB, thmbPos1,
thmbPos2]
        []      thmbPos2?z:pnt[z=x]; thmbPos1?y:pnt[x=y]; demux3[oaTHMB, thmbPos1,
thmbPos2])
endproc (* an AND behaviour *)

process demux4[d,d1,d2]: noexit :=
        d?x:int;
                (       d1?y:int[x=y]; d2?z:int[z=x]; demux4[d,d1,d2]
                []      d2?z:int[z=x]; d1?y:int[x=y]; demux4[d,d1,d2])
        []
        d?x:segment; d1?y:segment [x=y]; demux4[d,d1,d2]
endproc (* an AND behaviour *)

(*******************playPauseACU *******************************)
process playPauseACU[s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]: noexit
:=
        au[playRT, actionPlay](defaultRate)
        |[playRT, actionPlay]|
        controller[s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
where (* acu *)

process controller [s, su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll] : exit :=

s; run [su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]

where (* controller *)

process run[su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]: exit :=
        constraints[playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
                [>
        suspend [su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]
where  (* run *)

process constraints[playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                        pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll]:noexit :=

(   (playDSP; exit [] playBT; exit []  playRT?x:int; exit [] option; playBT; setAll;
exit) >>
        (actionPlay?y:int;    (iaBT; iaDSP; exit [] iaDSP; iaBT; exit) >>
                (       changesOfRate[playRT, actionPlay]
                [>
                        (choice B in [pauseDSP, pauseBT, pauseKbd, pauseBF]
                                [] B; actionPause; (iaBT; iaDSP; exit [] iaDSP; iaBT;
exit))))
 []
```

```
      pauseBF; exit )
(*if not playing ignore a pause and ignore options directed to the monitor *)

>>
constraints[playDSP, playBT, playRT, actionPlay, actionPause, pauseDSP, pauseBT,
pauseKbd,
                                    pauseBF, iaDSP, iaBT, option, setAll]
endproc

process changesOfRate[p,a] : noexit :=
      p?x:int; a?y:int; changesOfRate[p,a]
endproc

process suspend [su, ab, playDSP, playBT, playRT, actionPlay, actionPause,  pauseDSP,
                  pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option, setAll] : exit :=
            su;    (su; run[su, ab, playDSP, playBT, playRT, actionPlay, actionPause,
pauseDSP,
                                    pauseBT, pauseKbd, pauseBF, iaDSP, iaBT, option,
setAll]
                  []  ab; exit)
            [] ab; exit
endproc (* suspend *)
endproc (* run *)
endproc (* controller *)

process au[ia, oa](r:int):noexit :=
      ia?x:int;     au[ia, oa](x)
[]    oa?!r; au[ia, oa](r)
endproc (* au *)
endproc (* playPauseCU *)


(********************* volume ************************************)

process volume[s, su, ab, press, move, release, out, iaR,iaV, oa] : noexit:=
      adu[press, move, release, out, iaR, iaV, oa](preferredVolume,defaultVolumeBar,
defaultVolumeBar)
            |[press, move, release, out, iaR, iaV, oa]|
      controller [s, su, ab, press, move, release, out, iaR, iaV, oa]
where (* behaviour *)

process controller [s, su, ab, press, move, release, out, iaR, iaV, oa]:exit :=
      s; run [su, ab, press, move, release, out, iaR, iaV, oa]
where (* controller *)

process run[su, ab, press, move, release, out, iaR, iaV, oa]: exit :=
      constraints[press, move, release, out, iaR, iaV, oa]
            [>
      suspend [su, ab, press, move, release, out, iaR, iaV, oa]
where  (* run *)

process constraints[press, move, release, out, iaR, iaV, oa] :noexit:=
      triggers[press, move, release, out, iaR, iaV, oa]
            |[press, move, release]|
      inp_control[press, move, release]
endproc (* constraints *)

process triggers[press, move, release, out, iaR, iaV, oa]: noexit :=
      press; iaV?x:Int; out?x:volumeBar;triggers[press, move, release, out, iaR, iaV,
oa] []
      move?x:pnt; out?x:volumeBar; oa?x:Int;  triggers[press, move, release, out, iaR,
iaV, oa] []
      release; out?x:volumeBar;          triggers[press, move,release,out, iaR, iaV,
oa]  []
      iaR?x:rct;                         triggers[press, move, release, out, iaR, iaV,
oa] []
      out?x:volumeBar;                   triggers[press, move, release, out, iaR, iaV,
oa]
endproc
```

```
process inp_control[press, move, release]: noexit :=
      press; (repeat[move][>release; inp_control[press, move, release])
endproc

process repeat[M]: noexit :=
      M?x:pnt;repeat[M]
endproc

process suspend [su, ab, press, move, release, out, iaR, iaV, oa]: exit :=
            su;    (su; run[su, ab, press, move, release, out, iaR, iaV, oa]
                   []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[press, move, release,  out, iaR, iaV, oa](a:Int,dc,ds:volumeBar) : noexit :=
      out!dc;       adu[press, move, release, out, iaR, iaV, oa] (a, dc, dc) []
      iaR?x:rct;    adu[press, move, release, out, iaR, iaV, oa] (receiveRV(a,x),
renderRV(dc,x), ds) []
      iaV?x:Int;    adu[press, move, release, out, iaR, iaV, oa] (receiveV(a,x),
renderV(dc,x), ds) []
      press;        adu[press, move, release, out, iaR, iaV, oa] (inputPr(ds,a),
echoPr(ds,a),ds) []
      move?x:pnt;   adu[press, move, release, out, iaR, iaV, oa] (inputMov(x,dc,a),
echoMov(x,dc,a), ds) []
      release;      adu[press, move, release, out, iaR, iaV, oa]
(inputRel(ds,a),echoRel(ds,a), ds) []
      oa!result(a);adu[press, move, release, out, iaR, iaV, oa] (a,echoRel(ds,a), ds)
endproc (*volume_ADU  *)

endproc (* volumeControl *)


(**************  playPauseButton ********************)
process playPauseButton[s, su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play,
pause] : noexit :=
      du[press, moveIn, moveOut, release, out, iaGC, ia](play_icon, play_icon)
            |[press, moveIn, moveOut, release, out, iaGC, ia]|
      controller [s, su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play,
pause]
where (* behaviour *)

process controller [s, su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play,
pause] : exit :=
      s; run [su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
where (* controller *)

process run[su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play, pause] : exit
:=
      constraints[press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
            [>
      suspend [su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
where  (* run *)

process constraints[press, moveIn, moveOut, release, out, iaGC, ia, play, pause] :
noexit :=
      ((inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
            |[press, moveIn, moveOut, release]|
      inp[press, moveIn, moveOut, release])
            |[press, moveIn, moveOut, play, pause]|
      triggers[press, moveIn, moveOut, play, pause])
            |[play, ia, pause]|
      toggle[play, ia, pause]
endproc (* constraints *)

process triggers[press, moveIn, moveOut, play, pause]: noexit :=
```

```
      (choice G1 in [press, moveIn, moveOut]
            [] G1; (choice G2 in [play, pause] [] G2;
                  triggers[press, moveIn, moveOut, play, pause]))
endproc

process inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play, pause]: noexit :=
        (choice G in [press, moveIn, moveOut, release, ia] [] G;
out?x:twoStateButton_dsp;
            inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play,pause])
      [] iaGC?x:rct; out?x:twoStateButton_dsp;
         inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play, pause]
      [] (choice G2 in [play, pause] [] G2; ia;
         inp_out[press, moveIn, moveOut, release, out, iaGC, ia, play, pause])

endproc (* inp_out *)

process inp[press, moveIn, moveOut, release]: noexit :=
      press; (repeat2[moveIn, moveOut][>
            release; inp[press, moveIn, moveOut, release])
endproc

process repeat2[moveIn, moveOut]:noexit :=
      moveOut; moveIn; repeat2[moveIn, moveOut]
endproc

process toggle[A, ia, B]: noexit :=
      A; ia; toggle[B, ia, A]
      []
      ia; toggle[B, ia, A]
endproc (* toggle *)

process suspend [su, ab, press, moveIn, moveOut, release, out, iaGC, ia, play, pause]:
exit :=
            su;    (su; run[su, ab, press, moveIn, moveOut, release, out, iaGC, ia,
play, pause]
                  []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process du[press, moveIn, moveOut, release,  out, iaGC, ia](dc,ds: twoStateButton_dsp) :
noexit :=
      out!dc; du[press,moveIn, moveOut, release, out, iaGC, ia] (dc, dc) []
      iaGC?x:rct; du[press,moveIn, moveOut, release, out, iaGC, ia] (renderR(dc,x), ds)
[]
      ia; du[press,moveIn, moveOut, release, out, iaGC, ia] (renderS(dc), ds) []
      press; du[press, moveIn, moveOut, release, out, iaGC, ia] (echoPr(dc), ds) []
      moveIn;  du[press, moveIn, moveOut, release, out, iaGC, ia] (echoMovIn(dc), ds)
[]
      moveOut; du[press, moveIn, moveOut, release, out, iaGC, ia] (echoMovOut(dc), ds)
[]
      release; du[press, moveIn, moveOut, release, out, iaGC, ia] (echoRel(ds), ds)
endproc (* playPause_dU  - a no abstraction ADU  *)

endproc(* playPause_dcu*)

(*********** DSP interactor ***********************************)
process dsp[s, su, ab, click, doubleClick, out, iaDSP, iaGcDsp, iaV, ok, play, ps]
:noexit :=
      adu [out, iaGcDsp, iaV](movieDisplay, movieDisplay)
            |[out, iaGcDsp, iaV]|
      controller [s, su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play,
ps]
where (* interaction *)

process controller [s, su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play,
ps] : exit :=
```

```
      s; run [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
where (* controller *)

process run[su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]: exit :=
      (constraints[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
            [>
      suspend [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps])
where  (* run *)

process constraints[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] :
noexit :=
      triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
      |[iaDsp, click, doubleClick]|
      toggle[iaDsp, doubleClick, click]
endproc

process triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] :noexit :=
      click;  ps;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      doubleClick; play;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaGcDsp?x:rct;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaV?x:frame; out?x:disp; ok;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaV?x:still; out?x:disp;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaDsp;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
endproc (*triggers *)

process toggle [iaDsp, A, B] : noexit :=
      A; iaDsp;     toggle[iaDsp, B, A]
[]    iaDsp;        toggle[iaDsp, B, A]
endproc

process suspend [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]:
exit :=
            su;(su; run[su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok,
play, ps]
                  []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[out,iaR, iaV](dc,ds: disp) : noexit :=
      out!dc;       adu[out, iaR, iaV](dc, dc) []
      iaR?x:rct;    adu[out, iaR, iaV](renderR(dc,x), ds) []
      iaV?x:frame;  adu[out, iaR, iaV](renderF(dc,x), ds) []
      iaV?x:still;  adu[out, iaR, iaV](renderS(dc,x), ds)
endproc (* adu *)

endproc (* disp *)

(*************** Functional Core ***************)
process fnctCore[actionPlay, actionPause,  actionGetMovieTime, out_ok,
      actionGotoTime,gotoBeginningOfMovie,gotoEndOfMovie,   actionSetAllFrames,
actionSetSelection, actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video, data, init]: noexit :=

      init ?M:movie;

      videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
      actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
            actionSetAllFrames,actionSetSelection,actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video, data, init](M)
```

```
      s; run [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
where (* controller *)

process run[su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]: exit :=
      (constraints[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
            [>
      suspend [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps])
where  (* run *)

process constraints[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] :
noexit :=
      triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
      |[iaDsp, click, doubleClick]|
      toggle[iaDsp, doubleClick, click]
endproc

process triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] :noexit :=
      click;  ps;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      doubleClick; play;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaGcDsp?x:rct;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaV?x:frame; out?x:disp; ok;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaV?x:still; out?x:disp;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps] []
      iaDsp;
            triggers[click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]
endproc (*triggers *)

process toggle [iaDsp, A, B] : noexit :=
      A; iaDsp;     toggle[iaDsp, B, A]
[]    iaDsp;        toggle[iaDsp, B, A]
endproc

process suspend [su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok, play, ps]:
exit :=
            su;(su; run[su, ab, click, doubleClick, out, iaDsp, iaGcDsp, iaV, ok,
play, ps]
                  []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[out,iaR, iaV](dc,ds: disp) : noexit :=
      out!dc;       adu[out, iaR, iaV](dc, dc) []
      iaR?x:rct;    adu[out, iaR, iaV](renderR(dc,x), ds) []
      iaV?x:frame;  adu[out, iaR, iaV](renderF(dc,x), ds) []
      iaV?x:still;  adu[out, iaR, iaV](renderS(dc,x), ds)
endproc (* adu *)

endproc (* disp *)

(*************** Functional Core ***************)
process fnctCore[actionPlay, actionPause,  actionGetMovieTime, out_ok,
      actionGotoTime,gotoBeginningOfMovie,gotoEndOfMovie,   actionSetAllFrames,
actionSetSelection, actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video, data, init]: noexit :=

      init ?M:movie;

      videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
      actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
            actionSetAllFrames,actionSetSelection,actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video, data, init](M)
```

```
endproc

process videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
        out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video,data, init]
        (M:movie):noexit :=

    actionPlay?rt:Int [0 lt rt];
        playF[actionPlay, actionPause,  actionGetMovieTime,
        out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video,data, init]
        (true, setRate(M,rt))
    []
    actionPlay?rt:Int[rt lt 0];
        playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
            actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection,actionSetVolume,
            actionGetMovieVolume, mcSetMovieBox, video, data, init]
            (true, setAllFrames(setRate(M,rt),false))
    []
    actionPlay?rt:Int [rt eq 0] ;
        video !getMoviePict(M);
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
            out_ok, actionGotoTime, gotoBeginningOfMovie,
        gotoEndOfMovie, actionSetAllFrames,
                actionSetSelection, actionSetVolume,
            actionGetMovieVolume, mcSetMovieBox, video, data,
    init](setRate(M,0))
    []
    actionGetMovieTime;
    data !movieTime(M);
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames,
        actionSetSelection, actionSetVolume, actionGetMovieVolume,
    mcSetMovieBox, video, data, init](M)
    []
    actionGotoTime ?targetPosition:Int;
    video !getMoviePict(setMovieTime(M,targetPosition));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setMovieTime(M, targetPosition))
    []
    gotoBeginningOfMovie;
    video !getMoviePict(setMovieTime(M,0));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setMovieTime(M, 0))
    []
    gotoEndOfMovie;
    video !getMoviePict(setMovieTime(M,duration(M)));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setMovieTime(M, duration(M)))
    []
    actionSetSelection ?selection:segment;
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setSelection(M, selection))
```

```
        []
        actionSetAllFrames;
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames,
              actionSetSelection, actionSetVolume, actionGetMovieVolume,
        mcSetMovieBox, video, data, init] (setAllFrames(M, true))
        []
        actionSetVolume ?newVolumeLevel:Int;
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
              (setVolume(M,newVolumeLevel))
              []
        actionGetMovieVolume !volume(M);
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init](M)
              []
        mcSetMovieBox?R:rct;
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection,
        actionSetVolume, actionGetMovieVolume, mcSetMovieBox,              video,
data, init] (setMovieBox(M,R))
endproc

process playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
              (ready:Bool, M:movie): noexit :=
        [ready and (movieTime(M) le duration(M))] ->
              video !videoFrame(M);
              data !movieTime(M);
              playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
              (false, incr(M))
              []
        [ready and (movieTime(M) eq duration(M))] ->
              video !videoFrame(M);
              out_ok;
              data !movieTime(M);
              videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
                    out_ok,actionGotoTime, gotoBeginningOfMovie,
              gotoEndOfMovie, actionSetAllFrames,
                          actionSetSelection, actionSetVolume,
                    actionGetMovieVolume, mcSetMovieBox, video, data,
        init](setAllFrames(M,false))
              []
        [ready]->i;
        (hide missingFrame in
        ([not(allFrames(M))] ->missingFrame;
              playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume,                    mcSetMovieBox, video,
        data, init] (ready, incr(M))
              []
        [allFrames(M)] ->missingFrame;
              playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
                    actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames,actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
                    (ready, M)))
```

Formal Specification of the Simple Player ™ graphical interface using the ADC interactor model .

```
        []
[not(ready)] -> i;
playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames,
        actionSetSelection, actionSetVolume, actionGetMovieVolume,
mcSetMovieBox, video, data, init](ready, M)
        []
actionGetMovieTime;
data !movieTime(M);
playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data,
init](ready, M)
        []
actionGotoTime ?targetPosition:Int;
video !getMoviePict(setMovieTime(M,targetPosition));
videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setMovieTime(M, targetPosition))
        []
gotoBeginningOfMovie;
video !getMoviePict(setMovieTime(M,0));
videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (setMovieTime(M, 0))
        []
gotoEndOfMovie;
video !getMoviePict(setMovieTime(M,duration(M)));
videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames,
        actionSetSelection, actionSetVolume, actionGetMovieVolume,
mcSetMovieBox, video, data, init]
        (setMovieTime(M, duration(M)))
        []
out_ok;
playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data, init]
        (true, M)
        []
actionSetVolume ?newVolumeLevel:Int;
playF[actionPlay, actionPause,  actionGetMovieTime,
        out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data, init]
                        (ready, setVolume(M,newVolumeLevel))
        []
actionGetMovieVolume !volume(M);
playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video, data,
init](ready, M)
        []
actionPlay?rt:Int [0 lt rt];
playF[actionPlay, actionPause,  actionGetMovieTime,
        out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
actionSetAllFrames, actionSetSelection, actionSetVolume,
actionGetMovieVolume, mcSetMovieBox, video,data, init](true,
setRate(M,rt))
        []
```

```
        actionPlay?rt:Int [rt lt 0];
              playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
                    actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
                    actionSetAllFrames,actionSetSelection, actionSetVolume,
                    actionGetMovieVolume, mcSetMovieBox, video, data, init]
                    (true, setRate(M,rt))
              []
        actionPlay?rt:Int [rt eq 0];
              video !getMoviePict(M);
              videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
                    out_ok, actionGotoTime, gotoBeginningOfMovie,
              gotoEndOfMovie, actionSetAllFrames,
                          actionSetSelection, actionSetVolume,
                    actionGetMovieVolume, mcSetMovieBox, video, data,
        init] (setAllFrames(setRate(M,0),false))
              []
        actionPause;
              video !getMoviePict(M);
              videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
              out_ok,actionGotoTime, gotoBeginningOfMovie,
              gotoEndOfMovie, actionSetAllFrames, actionSetSelection,
              actionSetVolume, actionGetMovieVolume, mcSetMovieBox,
              video, data, init]

        (setAllFrames(setRate(M,0),false))
              []
        mcSetMovieBox?r:rct;
        playF[actionPlay, actionPause,  actionGetMovieTime, out_ok,
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
              (ready, setMovieBox(M,r))
endproc

process playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init]
                                    (ready:Bool,M:movie):noexit:=
        [ready and (duration(M) lt movieTime(M))] ->
        video !videoFrame(M);
        data !movieTime(M);
        playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data,
        init](false, decr(M))
                    []
        [ready and (movieTime(M) eq  duration(M))] ->
        video !videoFrame(M);
        out_ok;
        data !movieTime(M);
        videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data, init](M )
                    []
        [ready]-> i;
        (hide missingFrame in
        ([not(allFrames(M))] -> missingFrame;
              playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
              actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
        actionSetAllFrames, actionSetSelection, actionSetVolume,
        actionGetMovieVolume, mcSetMovieBox, video, data,
        init](ready, decr(M))
                    []
        [allFrames(M)] -> missingFrame;
              playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
```

```
        actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data,
    init](ready, M)))
                  []
    [not(ready)] -> i;
    playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data,
    init](ready, M)
                  []
    actionGetMovieTime;
    data !movieTime(M);
    playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data,
    init](ready, M)
                  []
    actionGotoTime ?targetPosition:Int;
    video !getMoviePict(setMovieTime(M,targetPosition));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
           (setMovieTime(M, targetPosition))
                  []
    gotoBeginningOfMovie;
    video !getMoviePict(setMovieTime(M,0));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
           (setMovieTime(M, 0))
                  []
    gotoEndOfMovie;
    video !getMoviePict(setMovieTime(M,duration(M)));
    videoPlayer[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
           (setMovieTime(M, duration(M)))
                  []
    out_ok;
    playB[actionPlay, actionPause,  actionGetMovieTime,
           out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
           (true, M)
                  []
    actionSetVolume ?newVolumeLevel:Int;
    playB[actionPlay, actionPause,  actionGetMovieTime,
           out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data, init]
           (ready, setVolume(M,newVolumeLevel))
           []
    actionGetMovieVolume !volume(M);
    playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
           actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
    actionSetAllFrames, actionSetSelection, actionSetVolume,
    actionGetMovieVolume, mcSetMovieBox, video, data,
    init](ready, M)
           []
    actionPlay?rt:Int [0 lt rt];
    playF[actionPlay, actionPause,  actionGetMovieTime,
```

```
            out_ok,actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
      actionSetAllFrames, actionSetSelection, actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video,data, init]
            (true, setRate(M,rt))
            []
      actionPlay?rt:Int [rt lt 0];
            playB[actionPlay, actionPause,  actionGetMovieTime, out_ok,
                  actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
      actionSetAllFrames, actionSetSelection,
      actionSetVolume, actionGetMovieVolume, mcSetMovieBox,              video,
data, init](true, setRate(M,rt))
            []
      actionPlay?rt:Int [rt eq 0];
            video !getMoviePict(M);
            videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
                  out_ok,actionGotoTime, gotoBeginningOfMovie,
            gotoEndOfMovie, actionSetAllFrames,
                        actionSetSelection, actionSetVolume,
                  actionGetMovieVolume, mcSetMovieBox, video, data,
      init] (setAllFrames(setRate(M,0),false))
            []
      actionPause;
            video !getMoviePict(M);
            videoPlayer[actionPlay, actionPause,  actionGetMovieTime,
            out_ok,actionGotoTime, gotoBeginningOfMovie,
      gotoEndOfMovie, actionSetAllFrames,
                  actionSetSelection, actionSetVolume,
            actionGetMovieVolume, mcSetMovieBox, video, data, init]

      (setAllFrames(setRate(M,0),false))
            []
      mcSetMovieBox?r:rct;
      playB[actionPlay, actionPause, actionGetMovieTime, out_ok,
      actionGotoTime, gotoBeginningOfMovie, gotoEndOfMovie,
      actionSetAllFrames, actionSetSelection, actionSetVolume,
      actionGetMovieVolume, mcSetMovieBox, video, data, init]
            (ready, setMovieBox(M,r))
endproc

(***************  PLAYER ****************************************)
process plr[s, su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s] : noexit :=
      adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            (initPlayBarData, thePlayerBar, thePlayerBar)
            |[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]|
      controller [s, su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
where (* scr *)

process controller[s, su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]:exit :=
      s; run [su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
where (* controller *)

process run[su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]:exit :=
      constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            [>
      suspend [su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
where (* run *)

process constraints [inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s] : noexit :=
      iaRct?x:rct; constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            []
      iaSg?x:segment; constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            []
      iaNt?x:Int;  out?y:playbar; constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt,
oa_t, oa_s]
            []
      out?x:playbar;      constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s]
            []
      inp_m?x:pnt; oa_t?y:int;
```

```
            constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            []
      inp_mk?x:line;      oa_s?y:segment;
            constraints[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
endproc (* constraints *)

process suspend [su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]: exit :=
            su;    (su; run[su, ab, inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s]
                        []  ab; exit)
            [] ab; exit
endproc (* suspend *)
endproc (* run *)
endproc (* controller plr *)

process adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s]
            (a:playBarData, dc, ds: playbar) : noexit :=
      inp_m?x:pnt; adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s](inputM(x,ds,a),echoM(x,ds,a),ds) []
      inp_mk?x:line; adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s](inputMK(x,ds,a),echoMK(x,ds,a),ds) []
      oa_t!resultT(a); adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s](a, dc,
ds) []
      oa_s!resultS(a); adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s](a, dc,
ds) []
      out!dc; adu [inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t, oa_s](a, dc, dc) []
      iaRct?x:rct; adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s](receiveRct(a,x),renderRct(dc,x), ds)[]
      iaSg?x:segment; adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s](receiveS(a,x),renderS(dc,x), ds)[]
      iaNt?x:Int; adu[inp_m, inp_mk, out, iaRct, iaSg, iaNt, oa_t,
oa_s](receiveT(a,x),renderT(dc,x), ds)
endproc (* plr adu *)
endproc (* plr  *)

(*********** selectionBand *****************)
process selectionBand[s, su, ab, inpPnt, out, ia, oa, erase, enable, send]:noexit:=
      adu[inpPnt, out, ia, oa, erase](noSelection, thePlayerBar, thePlayerBar)
            |[inpPnt, out, ia, oa, erase]|
      controller [s, su, ab, inpPnt, out, ia, oa, erase, enable, send]

where (* behaviour *)

process controller [s, su, ab, inpPnt, out, ia, oa, erase, enable, send] : exit :=
      s; run [su, ab, inpPnt, out, ia, oa, erase, enable, send]
where (* controller *)

process run[su, ab, inpPnt, out, ia, oa, erase, enable, send] : exit :=
      (constraints[inpPnt, out, ia, oa, erase, enable, send]
            [>
      suspend [su, ab, inpPnt, out, ia, oa, erase, enable, send])
where  (* run *)

process constraints[inpPnt, out, ia, oa, erase, enable, send] : noexit :=
      (enable; (operation[inpPnt, out, ia, oa, erase, send]
      [> send; oa?x:line; constraints[inpPnt, out, ia, oa, erase, enable, send]))
[]    ia?x:playBar;constraints[inpPnt, out, ia, oa, erase, enable, send]
[]    out?x:playBar; constraints[inpPnt, out, ia, oa, erase, enable, send]
[]    erase; out?x:playBar;oa?x:line;
                  constraints[inpPnt, out, ia, oa, erase, enable, send]
endproc  (* constraints *)

process operation[inpPnt, out, ia, oa, erase, send] : noexit :=
      inpPnt?x:pnt; out?x:playBar; operation[inpPnt, out, ia, oa, erase, send]
[]    out?x:playBar; operation[inpPnt, out, ia, oa, erase, send]
endproc (* operation *)

process suspend [su, ab, inpPnt, out, ia, oa, erase, enable, send]: exit :=
            su;    (su; run[su, ab, inpPnt, out, ia, oa, erase, enable, send]
```

```
                    []  ab; exit)
              [] ab; exit
endproc (* suspend *)
endproc (* run *)
endproc (* controller *)

process adu[inpPnt, out, ia, oa, erase] (a:line,pc,ps:playBar) : noexit :=
      oa!a;          adu[inpPnt, out, ia, oa, erase] (a,pc,ps) []
      out!pc;        adu[inpPnt, out, ia, oa, erase] (a,pc,pc) []
      ia?x:playBar;adu[inpPnt, out, ia, oa, erase] (receivePB(a,x),renderPB(pc,x),ps)
[]
      inpPnt?x:pnt;adu[inpPnt, out, ia, oa, erase] (inputPnt(x,a),echo(x,ps,a),ps) []
      erase;         adu[inpPnt, out, ia, oa, erase] (inputEr(a),echoEr(pc, a),ps)
endproc (* DRAG_AU *)

endproc (* selectionBand *)

(************* XCONTROLLER ********************************)
process xcontroller[s, su, ab, kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
      relPLR, pressFwd, relFwd, pressBwd, relBwd,
      pressRate, relRate, pauseKbd, enable, send, erase] : exit :=

      s; run [su, ab, kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
            relPLR, pressFwd, relFwd, pressBwd, relBwd,
            pressRate, relRate, pauseKbd, enable, send, erase]
where (* xcontroller *)

process run[su, ab, kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
      relPLR, pressFwd, relFwd, pressBwd, relBwd,
      pressRate, relRate, pauseKbd, enable, send, erase] : exit :=

      constraints[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
            relPLR, pressFwd, relFwd, pressBwd, relBwd,
            pressRate, relRate, pauseKbd, enable, send, erase]
                  [>
      suspend [su, ab, kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
            relPLR, pressFwd, relFwd, pressBwd, relBwd,
            pressRate, relRate, pauseKbd, enable, send, erase]
endproc (* run *)

process constraints[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
      relPLR, pressFwd, relFwd, pressBwd, relBwd,
      pressRate, relRate, pauseKbd, enable, send, erase]: noexit :=


(     kbdSftPress;  kbdSftMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
                  relPLR, pressFwd, relFwd, pressBwd, relBwd,
                  pressRate, relRate, pauseKbd, enable, send, erase]
[]    pressPLR?x:pnt; erase;     (exit []
                        kbdSftPress; enable;
                              (kbdSftRel; send; exit []
                              relPLR?x:pnt; send; kbdSftMode[kbdSftPress,
kbdSftRel, pressBT, pauseBT,
                                                pressPLR, relPLR, pressFwd, relFwd,
pressBwd, relBwd,
                                                pressRate, relRate, pauseKbd, enable,
send, erase]))
[]    pressRate; kbdCtrlMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
            relPLR, pressFwd, relFwd, pressBwd, relBwd,
            pressRate, relRate, pauseKbd, enable, send, erase]
[]    (choice x in [pressBT, pauseBT, pressFwd, relFwd, pressBwd, relBwd] []x; exit) )
      (* the last clause will unconstrain other sequences of its gates *)

>>
      constraints[kbdSftPress,kbdSftRel,pressBT, pauseBT, pressPLR,relPLR, pressFwd,
relFwd,
                  pressBwd,  relBwd, pressRate, relRate, pauseKbd, enable, send,
erase]
```

```
endproc

process kbdSftMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
              relPLR, pressFwd, relFwd, pressBwd, relBwd,
              pressRate, relRate, pauseKbd, enable, send, erase]:noexit :=

       kbdSftRel; exit
[]     pressBT; enable;
              (kbdSftRel; send; exit
              []
              pauseBT; send;
              kbdSftMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
                     relPLR, pressFwd, relFwd, pressBwd, relBwd,
                     pressRate, relRate, pauseKbd, enable, send, erase])
[]     pressPLR?x:pnt; enable;
              (relPLR?x:pnt; send; kbdSftMode[kbdSftPress, kbdSftRel, pressBT, pauseBT,
pressPLR,
                     relPLR, pressFwd, relFwd, pressBwd, relBwd,
                     pressRate, relRate, pauseKbd, enable, send, erase]
              []
              kbdSftRel; send; exit)
[]     pressFwd; enable; relFwd; send; exit
[]     pressBwd; enable; relBwd; send; exit
[]     pressRate; enable; (choice X in [kbdSftRel, pauseBT]
              [] X; send; kbdCtrlMode[kbdSftPress, kbdSftRel, pressBT, pauseBT,
pressPLR,
                           relPLR, pressFwd, relFwd, pressBwd, relBwd,
                           pressRate, relRate, pauseKbd, enable, send, erase])
>>
       constraints[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
              relPLR, pressFwd, relFwd, pressBwd, relBwd,
              pressRate, relRate, pauseKbd, enable, send, erase]

endproc  (* kbdSftMode *)

process kbdCtrlMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
              relPLR, pressFwd, relFwd, pressBwd, relBwd,
              pressRate, relRate, pauseKbd, enable, send, erase]:noexit :=
       relRate; constraints[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
              relPLR, pressFwd, relFwd, pressBwd, relBwd,
              pressRate, relRate, pauseKbd, enable, send, erase]
[]     kbdSftPress; enable; (pauseBT; send;
                     kbdSftMode[kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
                            relPLR, pressFwd, relFwd, pressBwd, relBwd,
                            pressRate, relRate, pauseKbd, enable, send, erase]
                     []
                     kbdSftRel; send; pauseKbd; kbdCtrlMode[kbdSftPress, kbdSftRel,
pressBT, pauseBT, pressPLR,
                                   relPLR, pressFwd, relFwd, pressBwd, relBwd,
                                   pressRate, relRate, pauseKbd, enable, send, erase])
endproc

process suspend [su, ab, kbdSftPress, kbdSftRel, pressBT, pauseBT, pressPLR,
                     relPLR, pressFwd, relFwd, pressBwd, relBwd,
                     pressRate, relRate, pauseKbd, enable, send, erase] : exit :=
              su;    (su; constraints[kbdSftPress, kbdSftRel, pressBT, pauseBT,
pressPLR,
                            relPLR, pressFwd, relFwd, pressBwd, relBwd,
                            pressRate, relRate, pauseKbd, enable, send, erase]
                     []  ab; exit)
              [] ab; exit
endproc (* suspend *)
endproc (* xcontroller *)


(******************* THUMB *********************)
process thumb[s, su, ab, press, move, release, out, ia, oa]: noexit :=

adu[press, move, release, out, ia, oa](indicator, thumb, thumb)
```

```
                |[press, move, release, out, ia, oa]|
controller [s, su, ab, press, move, release, out, ia, oa]
where

process controller [s, su, ab, press, move, release, out, ia, oa] : exit :=
       s; run [su, ab, press, move, release, out, ia, oa]
where (* controller *)

process run[su, ab, press, move, release, out, ia, oa] : exit :=
       (constraints[press, move, release, out, ia, oa]
              [>
       suspend [su, ab,press, move, release, out, ia, oa])
where  (* run *)

process constraints[press, move, release, out, ia, oa] : noexit :=
       inp[press, move, release]
              |[press, move, release]|
       trigger[press, move, release, out, oa, ia]

endproc (* constraints *)

process trigger[press, move, release, out, oa, ia]: noexit :=
       (choice X in [press,move,release]
              []X?y:pnt; out?z:thumb_dsp; oa?q:pnt;
                           trigger[press, move, release, out, oa, ia])
       []
       (ia?x:playBar;
       out?z:thumb_dsp; trigger[press, move, release, out, oa, ia])
endproc(* trigger *)

process inp[press, move, release]: noexit :=
       press?x:pnt;
       (repeat[move]
              [>
       release ?x:pnt;
       inp[press, move, release])
endproc (* inp *)

process repeat[inp]:noexit :=
       inp?x:pnt; repeat[inp]
endproc (* repeat *)

process suspend [su, ab, press, move, release, out, ia, oa]: exit :=
              su;    (su; run[su, ab, press, move, release, out,
ia, oa]
                     []  ab; exit)
              [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[press, move, release, out, ia, oa]
              (a:pnt, dc,ds: thumb_dsp) : noexit :=
       oa!result(a); adu[press, move, release, out, ia, oa](a, dc, ds) []
       out!dc; adu[press, move, release, out, ia, oa](a, dc, dc) []
       ia?x:playBar; adu [press, move, release, out, ia, oa]
                           (receive(a,x),render(dc,x), ds) []
       press ?x:pnt; adu[press, move, release, out, ia, oa]
              (inputPress(x,ds,a),echoPress(x,ds,a), ds) []
       move ?x:pnt; adu[press, move, release, out, ia, oa]
              (inputMove(x,ds,a), echoMove(x,ds,a),ds) []
       release?x:pnt; adu[press, move, release, out, ia, oa]
              (inputRelease(x,ds,a),echoRelease(x,ds,a), ds)
endproc (* THUMB_ADU *)

endproc (* thumb *)

(******************* PUSH buttons (bwd and fwd) ***********************)
```

Formal Specification of the Simple Player ™ graphical interface using the ADC interactor model .

```
process pushButton[s, su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]
                                        (ds:twoStateButton_dsp) : noexit :=
      du[press, moveIn, moveOut, release, out, iaGC] (ds, ds)
            |[press, moveIn, moveOut, release, out, iaGC]|
      controller [s, su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]
where (* behaviour *)

process controller [s, su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]
: exit :=
      s; run [su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]
where (* controller *)

process run[su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause] : exit :=
      constraints[press, moveIn, moveOut, release, out, iaGC, goto, pause]
            [>
      suspend [su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]
where  (* run *)

process constraints[press, moveIn, moveOut, release, out, iaGC, goto, pause] : noexit :=
      (inp_out[press, moveIn, moveOut, release, out, iaGC]
            |[press, moveIn, moveOut, release]|
      inp[press, moveIn, moveOut, release, pause])
            |[press, moveIn, release, moveOut, iaGC, out]|
      triggers[press, moveIn, moveOut, release, goto, iaGC, out]
endproc (* constraints *)

process triggers[press, moveIn, moveOut, release, goto, iaGC, out]: noexit :=
      press; goto; triggers[press, moveIn, moveOut, release, goto, iaGC, out]
[]    moveIn; goto; triggers[press, moveIn, moveOut, release, goto, iaGC, out]
[]    moveOut; triggers[press, moveIn, moveOut, release, goto, iaGC, out]
[]    release; triggers[press, moveIn, moveOut, release, goto, iaGC, out]
[]    iaGC?x:rct; triggers[press, moveIn, moveOut, release, goto, iaGC, out]
[]    out?x:twoStateButton_dsp; triggers[press, moveIn, moveOut, release, goto, iaGC,
out]
(* the last four clauses ensure that the trigger process constrains the
interactor to perform a goto after a press or a moveIn.  Nevertheless,
inp_out does not offer pause, and pause is not in the set of sync
gates.  Thus it is not affected by this process. *)
endproc

process inp_out[press, moveIn, moveOut, release, out, iaGC]: noexit :=
        (choice G in [press, moveIn, moveOut, release] [] G; out?x:twoStateButton_dsp;
            inp_out[press, moveIn, moveOut, release, out, iaGC])
      [] iaGC?x:rct; out?x:twoStateButton_dsp;
         inp_out[press, moveIn, moveOut, release, out, iaGC]
      [] out?x:twoStateButton_dsp;
            inp_out[press, moveIn, moveOut, release, out, iaGC]
endproc (* inp_out *)

process inp[press, moveIn, moveOut, release, pause]: noexit :=
      press; pause; (repeat[press, moveIn, moveOut][>
            release; inp[press, moveIn, moveOut, release, pause])
endproc

process repeat[press, moveIn, moveOut]:noexit :=
      moveOut; moveIn; repeat[press, moveIn, moveOut]
[]    press; repeat[press, moveIn, moveOut]
endproc

process suspend [su, ab, press, moveIn, moveOut, release, out, iaGC, goto, pause]: exit
:=
            su;    (su; run[su, ab, press, moveIn, moveOut, release, out, iaGC,goto,
pause]
                   []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)
```

```
process du[press, moveIn, moveOut, release,  out, iaGC](dc,ds: twoStateButton_dsp) :
noexit :=
      out!dc;       du[press, moveIn, moveOut, release, out, iaGC] (dc, dc) []
      iaGC?x:rct;   du[press, moveIn, moveOut, release, out, iaGC] (renderR(dc,x), ds)
[]
      press;        du[press, moveIn, moveOut, release, out, iaGC] (echoPr(dc), ds) []
      moveIn;       du[press, moveIn, moveOut, release, out, iaGC] (echoMovIn(dc), ds)
[]
      moveOut;      du[press, moveIn, moveOut, release, out, iaGC] (echoMovOut(dc),
ds) []
      release;      du[press, moveIn, moveOut, release, out, iaGC] (echoRel(ds), ds)
endproc (* pushButton ADU  - a no abstraction ADU :  DU  *)

endproc(* pushButton_dcu*)
(******************  MONITOR Time ****************)
process monitor[s, su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia] : noexit :=
      au[oa_get, oa_goto, fwd, bwd, ia] (aTime)
            |[oa_get, oa_goto, fwd, bwd, ia]|
      controller [s, su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]
where (* behaviour *)

process controller [s, su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]: exit :=
      s; run [su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]
where (* controller *)

process run[su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]: exit :=
      constraints[option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]
            [>
      suspend [su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]
where  (* run *)

process constraints[option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]: noexit :=
      (choice x in [fwd, bwd] [] x; oa_get; ia?z:int; oa_goto?y:int;
            constraints[option, oa_get, oa_goto, fwd, jF, bwd, jB, ia])
      [] option;    (     fwd; jF; constraints[option, oa_get, oa_goto, fwd, jF, bwd,
jB, ia]
                    []    bwd; jB; constraints[option, oa_get, oa_goto, fwd, jF, bwd,
jB, ia])
endproc (* constraints *)


process suspend [su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]: exit :=
            su;   (su; run[su, ab, option, oa_get, oa_goto, fwd, jF, bwd, jB, ia]
                  []  ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process au[oa_get, oa_goto, fwd, bwd, ia](theTime:montr_abs): noexit :=
      oa_get;       au[oa_get, oa_goto, fwd, bwd, ia](theTime)
[]    ia?x:int;     au[oa_get, oa_goto, fwd, bwd, ia](receive(theTime,x))
[]    oa_goto !result(theTime);au[oa_get, oa_goto, fwd, bwd, ia](theTime)
[]    fwd;          au[oa_get, oa_goto, fwd, bwd, ia](inputF(theTime))
[]    bwd;          au[oa_get, oa_goto, fwd, bwd, ia](inputB(theTime))
endproc
endproc(* au *)

(**************  RESIZE BOX*****************************************)


process resize[s, su, ab, press, move, release, out, oa] : noexit:=
      adu[press, move, release, out, oa](graphicalContext, resizeBox, resizeBox)
            |[press, move, release, out, oa]|
      controller [s, su, ab, press, move, release, out, oa]
where (* behaviour *)
```

```
process controller [s, su, ab, press, move, release, out, oa]:exit :=
      s; run [su, ab, press, move, release, out, oa]
where (* controller *)

process run[su, ab, press, move, release, out, oa]: exit :=
      constraints[press, move, release, out, oa]
            [>
      suspend [su, ab, press, move, release, out, oa]
where  (* run *)

process constraints[press, move, release, out, oa] :noexit:=
      triggers[press, move, release, out, oa]
            |[press, move, release]|
      inp_control[press, move, release]
endproc (* constraints *)

process triggers[press, move, release, out, oa]: noexit :=
      press?x:pnt; out?x:pb_dsp;        triggers[press, move, release, out, oa] []
      move?x:pnt;  out?x:pb_dsp;        triggers[press, move, release, out, oa] []
      release?x:pnt;      out?x:pb_dsp; oa?x:rct; triggers[press, move, release,
out, oa] []
      out?x:rct;                        triggers[press, move, release, out, oa]
endproc

process inp_control[press, move, release]: noexit :=
      press?x:pnt; (repeat[move][>release?x:pnt; inp_control[press, move, release])
endproc

process repeat[M]: noexit :=
      M?x:pnt;repeat[M]
endproc

process suspend [su, ab, press, move, release, out, oa]: exit :=
            su;    (su; run[su, ab, press, move, release, out, oa]
                   [] ab; exit)
            [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[press, move, release,  out, oa](a:rct, dc,ds:pb_dsp) : noexit :=
      out!dc;       adu[press, move, release, out, oa] (a, dc, dc) []
      press?x:pnt; adu[press, move, release, out, oa] (inpPr(ds,a,x), echoPr(ds,a,x),
ds) []
      move?x:pnt;  adu[press, move, release, out, oa]
(inpMov(ds,a,x),echoMove(ds,a,x), ds) []
      release?x:pnt;      adu[press, move, release, out, oa] (inpRel(ds,a,x),
echoRel(ds,a,x), ds) []
      oa!resultRB(a);     adu[press, move, release, out, oa] (a, dc, ds)
endproc (*volume_ADU  *)
endproc (* resize box *)

(********************* rate ************************************)

process rate[s, su, ab, press, move, release, out, iaR, oa] : noexit:=
      adu[press, move, release, out, iaR, oa](defaultRate,defaultRateBar,
defaultRateBar)
            |[press, move, release, out, iaR, oa]|
      controller [s, su, ab, press, move, release, out, iaR, oa]
where (* behaviour *)

process controller [s, su, ab,  press, move, release, out, iaR, oa]:exit :=
      s; run [su, ab,  press, move, release, out, iaR, oa]
where (* controller *)

process run[su, ab,  press, move, release, out, iaR, oa]: exit :=
      constraints[press, move, release, out, iaR, oa]
            [>
```

```
      suspend [su, ab,  press, move, release, out, iaR, oa]
where  (* run *)

process constraints[press, move, release, out, iaR, oa] :noexit:=
      triggers[press, move, release, out, iaR, oa]
            |[press, move, release]|
      inp_control[press, move, release]
endproc (* constraints *)

process triggers[press, move, release, out, iaR, oa]: noexit :=
      press;       out?x:rateBar;      oa?x:Int;    triggers[press, move, release,
out, iaR, oa] []
      move?x:pnt;  out?x:rateBarpa?x:Int;    triggers[press, move, release, out,
iaR, oa] []
      release;     out?x:rateBar;  oa?x:Int;  triggers[press, move, release, out,
iaR, oa] []
      iaR?x:rct;                              triggers[press, move, release, out,
iaR, oa]
endproc

process inp_control[press, move, release]: noexit :=
      press; (repeat[move][>release; inp_control[press, move, release])
endproc

process repeat[M]: noexit :=
      M?x:pnt;repeat[M]
endproc


process suspend [su, ab,  press, move, release, out, iaR, oa]: exit :=
            su;    (su; run[su, ab,  press, move, release, out, iaR, oa]
                   [] ab; exit)
             [] ab; exit
endproc (* suspend *)

endproc (* run *)
endproc (* controller *)

process adu[press, move, release,  out, iaR, oa](a:Int,dc,ds:rateBar) : noexit :=
      out!dc;      adu[press, move, release, out, iaR, oa] (a, dc, dc) []
      iaR?x:rct;   adu[press, move, release, out, iaR, oa] (receiveRt(a,x),
renderRt(dc,x), ds) []
      press;       adu[press, move, release, out, iaR, oa] (inputPr(ds,a),
echoPr(ds,a),ds) []
      move?x:pnt;  adu[press, move, release, out, iaR, oa] (inputMov(x,dc,a),
echoMov(x,dc,a), ds) []
      release;     adu[press, move, release, out, iaR, oa]
(inputRel(ds,a),echoRel(ds,a), ds) []
      oa!result(a);adu[press, move, release, out, iaR, oa] (a,echoRel(ds,a), ds)
endproc (*rate_ADU  *)
endproc (* rateControl *)

endspec
```