ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

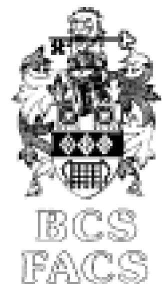
D. J. Duke , University of York, UK and A.S. Evans, University of Bradford, UK (Eds)

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, 14-15 July 1997

Using LOTOS for the Evaluation of Design Options in the PREMO Standard

G.P. Faconti and M. Massink



Published in collaboration with the British Computer Society

Springer

©*Copyright in this paper belongs to the author(s)*

Using LOTOS for the Evaluation of Design Options in the PREMO Standard

G. P. Faconti and M. Massink
CNR - Istituto CNUCE, Via S.Maria 36,
56126 PISA, Italy

Abstract

PREMO, Presentation Environment for Multimedia Objects, is a major new standard under development within ISO/IEC. It addresses the creation of, presentation of, and interaction with all forms of information using single or multiple media. The standard is written using an Object Oriented approach. In this paper we specify the behavioural aspects of one of the central objects of the standard, the PREMO synchronizable object, in a constraint oriented style. We show that by adding further constraints in a modular way various design options can be investigated by means of model checking. This provides designers with more reliable information concerning the behavioural aspects of different designs which obviously is helpful in the evaluation of design decisions. In this way formal methods do not only play a role in the final evaluation of the correctness of a design, but can be used in a design methodology in which a more dynamical process of design and evaluation is required. A requirement for this way of working is modularity which is well supported by both the Object Oriented and the Constraint Oriented approaches.

1 Introduction

Technology has evolved to the point that many interactive applications are enriched by the simultaneous use of multiple presentation media. While on the input side, the widespread use of sophisticated techniques, such as those employed within virtual reality systems, is limited to research laboratory prototypes, multimedia technology offers the capability of developing commercial products where the presentation becomes extremely complex in terms of the system architecture. We can think of systems as diverse as medical systems, real-time command control systems and geographical information systems. In each of these systems different presentation techniques may be used, such as the simultaneous use of 3D graphics, video animation, and sound. The presentation of data has become a complex task in which a large number of diverse requirements have to be taken into account.

In order to deal with this complexity in a way that allows for extension and adaptability to specific needs in a uniform way a standardization project is developing a standard, called PREMO [ISO96], that addresses the creation of, presentation of, and interaction with all forms of information using single or multiple media. In particular PREMO addresses the issues of configuration, extension, and interoperation of and between PREMO implementations. The approach taken within PREMO is an *object oriented* one. It defines complex object-oriented systems that are to be used in a distributed environment. Objects defined in PREMO are supposed to function largely independent and to cooperate and synchronize with other objects by means of communication. This aspect of PREMO requires a thorough investigation of the *behavioural aspects* of the combination of those objects.

Our goal is to get a deeper understanding of multimedia synchronization by specifying the behavioural aspects of one of the central objects in the standard: the synchronizable object. We based our specification on the english version of the standard, on a preliminary but quite detailed state-based specification [DDHF97], and on a previous process-algebraic specification [FM97]. Since we want to analyse the behavioural aspects

of independently operating objects that cooperate by means of communication, we use specification and abstraction techniques that have been developed within concurrency theory.

To this purpose, we want to investigate the use of Lotos (Language Of Temporal Ordering Specification) [ISO89], for the specification of the behavioural aspects of PREMO objects. We have already experimented with a new way to obtain a constraint oriented, modular Basic Lotos specification that can be directly used for computer assisted verification such as simulation and model checking. Our experience with this approach and some of the results obtained are recalled in sections 4 and 5 where we make use of the verification environments Lite (Lotos Integrated Tools Environment) [vEi91] and JACK [DFGR95, BFe96, BGL94]. The first provides tools for checking the syntax and some semantic properties of Lotos specifications and allows for simulation of the specification and its translation into a common format for automata. The latter can be used on this common format and gives an integrated set of tools which comprises (M)AUTO and Autograph [RDS90, MaV90] and the (X)AMC [DFGR91] model checker for action based branching-time temporal logic formulas.

The longterm aim of this research is to gain insights in the particular requirements for formalisms and tools for the analysis of specifications in the area of multimedia. The specification of PREMO Objects serves as a representative case study.

The paper is organized to give a short introduction to the PREMO standard with particular emphasis to the synchronization support mechanism in Section 2. This is followed by a short description of the tools making up our development environment in Section 3. Section 4 describes the basic specification of PREMO synchronizable objects and gives an analysis of its component processes. In Section 5 we refine the specification by describing the internal progression of synchronizable objects. In Section 6 we investigate different design options to define the *jump* operation that is used as an example to introduce a general methodology based on the constraint style of specification. In Section 7 we describe our main conclusions of this investigation and we give an outline of future research.

2 Short overview of the PREMO standard

PREMO, Presentation Environment for Multimedia Objects, is a major new standard under development in Joint Technical Committee 1 (JTC 1) of the International Organization for Standardization and the International Electrotechnical Committee (ISO/IEC) that addresses the creation of, presentation of, and interaction with all forms of information using single or multiple media. In particular, it addresses the issues of configuration, extension, and interoperation of and between PREMO implementations.

The aim of PREMO is the standardization of programming environments for the presentation of multimedia data. Construction of, and interaction with such multimedia information are also important aspects of PREMO; they are supported by application oriented modelling and interaction at several levels of abstraction. Multimedia is considered in a very general sense; high level virtual reality environments, which mix real time 3D rendering techniques with sound, video, or even tactile feedback, and their effects are, for example, within the scope of PREMO.

PREMO puts great emphasis on open environments which enable easy configurability, adaptability, and extendibility of standardized components. This way PREMO can react in a direct and timely manner to future developments and stay up to date over a longer period of time. The PREMO standard is organized as a set of largely separable components. Each component provides a set of functionality that is useful for a target application area.

An important aspect of PREMO is the use of object oriented techniques. The object model is a traditional one, and it is based on the concept of subtyping and inheritance. However, defining complex object-oriented systems to be used in a distributed environment leads to software engineering issues, whose complete solution would go far beyond the charter (and the expertise) of the PREMO working group. To this purpose, a Special Rapporteur has been appointed to provide a report on the applicability of formal description techniques to multi-media standards, with particular regard to formally specifying object behaviour and interfaces.

Following the recommendations of the Special Rapporteur's Report [RDD94], the Object-Z formal nota-

tion [DRS94], an object oriented extension of the Z notation [Spi89], has been used to specify the behaviour of PREMO objects themselves. The choice was largely determined by the available expertize in the PREMO Rapporteur Group and a certain affinity between the state-based nature of PREMO itself and the state-based description techniques.

One of the more challenging aspects that have been faced, is the specification of the PREMO synchronization model. Objects can communicate with one another through messages, i.e. through the operations defined on the object types. Objects can become suspended either by waiting for an operation invocation to return, or by waiting on the arrival of an operation request. Consequently, operations on objects serve as a vehicle to synchronize various activities. Whether the concurrent activity of active objects is realized through separate hardware processors, through distribution over a network, or through some multithreaded operating system, is not relevant to PREMO and it is considered to be an implementation dependency. The emphasis on the activity of objects stems from the need for synchronization in multimedia environments and forms the basis of the synchronization model described in this paper. Using concurrency to achieve synchronization in multimedia systems is not specific to PREMO. Other models and systems have taken a similar approach (see for example [DNNRA93]) and PREMO, whose task is to provide a synthesis for current standardization, has been obviously influenced by these models.

2.1 Supporting Synchronization in PREMO

The PREMO synchronization model is based on the fact that objects can be active elements. Different continuous media (e.g. a video sequence and corresponding sound track) are modelled as concurrent activities that may have to reach specific milestones at distinct and possibly user definable synchronization points. This is the event based synchronization approach, which forms the basic layer of synchronization in PREMO. Although a large number of synchronization tasks are, in practice, related to synchronization in time, the choice of an essentially timeless synchronization scheme offers greater flexibility. While time-related synchronization schemes can be built on top of an event-based synchronization model, it is sometimes necessary to support purely event-based synchronization to achieve special effects required by some application. Examples of how the various synchronization objects may be used can be found in [ISO96].

In line with the object-oriented approach of PREMO, the synchronization model defines abstract object types that capture the essential features of synchronization. For the event-based synchronization scheme two major object types are defined:

- synchronizable objects, which form the super types of, e.g., various media objects types;
- synchronization points, which may be used to manage complex synchronization patterns amongst synchronizable objects.

2.2 The PREMO Synchronizable Objects

A synchronizable object is a finite state machine modelling various operations that control the presentation of a medium within a possibly complex multi-media presentation environment.

A synchronizable object can be in one of four states or modes (namely STOPPED, PLAYING, PAUSING and WAITING). A number of operations are defined that result in state transitions, and that modify or inquire several parameters controlling the progression along an internal one dimensional co-ordinate space. No particular interpretation is placed on these states, except that certain operations can only be performed in certain states. Similarly, the internal co-ordinate space is introduced as an abstract concept to which different synchronizable objects can add a different semantic meaning. The intention is that object types representing different kinds of media will inherit from this class and specialise the co-ordinate space and state machine in an appropriate way. (i.e. media objects, such as audio, may represent time while others, such as video, may address frame numbers along this space).

Reference points can be defined on the internal co-ordinate space where synchronization elements can be attached. Synchronization elements contain information on an event instance, a reference to a PREMO

object, a reference to one of the operations of this object, and, finally, a Wait flag. When a reference point is reached during the progression, the synchronizable object uses the stored information to call, by means of messaging, the indicated operation on the referenced object using the event instance as an argument to the call. Finally it may suspend itself if the Wait flag is set to true. This mechanism allows for the definition of complex synchronization patterns amongst objects so that one synchronizable object can stop other objects, restart them, suspend them, etc.

Synchronizable objects can progress through the coordinate space in a certain direction by performing a sequence of progression stages. Each stage consists of the computation of a new required position and the stepwise traversal of the space between the current position and the new position by means of a separate pointer. In every step the information found at the coordinate is presented. If a reference point is encountered the call-back mechanism descibed above is activated. At the end of each stage the current position is updated. A stage is essentially atomic which means that external actions such as the mode transitions are delayed until an appropriate moment. Further details on progression are given in Section 5. A graphical presentation of the coordinate space and its relevant attributes and reference points are given in Figure 1.

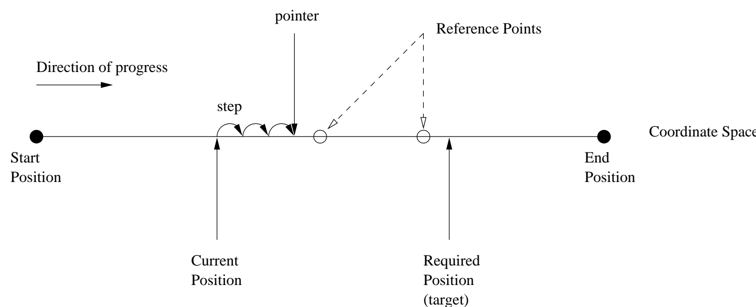


Figure 1: Simplified representation of coordinate space of synchronizable object

Finally synchronizable objects contain also operations to change the value of position pointers, to add and delete reference points, and to add and delete synchronization elements associated with reference points.

3 Development environment

We are interested in investigating the behaviour of synchronizable objects in greater detail than it is possible with a state based specification. The Object-Z specification of PREMO objects, accompanying the standard, explains and explores the state space of objects by defining the variables that make up this state, and the operations that result in changing the value of one or more of such variables. However, Object-Z, as any other state based specification such as for example Z and VDM [Bic94], doesn't fully exploit the temporal order relations between operations. In order to do this, a process algebraic approach is better suited and gives a complementary view on the system under specification. Here, Basic Lotos has been chosen out of other available notations, such as CSP [Hoa78] and CCS [Mil89], mainly because it is the only formal specification language that has been standardised so far, and we want to explore its applicability in the field of multimedia standards.

Our strategy of investigation is based on developing a textual Basic Lotos specification of the basic transitions of modes of PREMO synchronizable objects. This specification is directly derived from the original Object-Z one following an approach outlined in the next section. The initial specification is subsequently refined by adding further details (in this case, the progression along the internal co-ordinate space) described by means of separate processes that are composed in parallel with synchronization following a constraint oriented style of specification [VSv91]. Finally, we describe additional operations on synchronizable objects by using the same specification style. We specify each operation as a separate set of processes that is subsequently composed with, and consequently constrained by the already defined ones. This enables

modularity in the development process and encourages the use of formal methods to support the discussion about possible design solutions while preserving the capability of verification or calculation of properties.

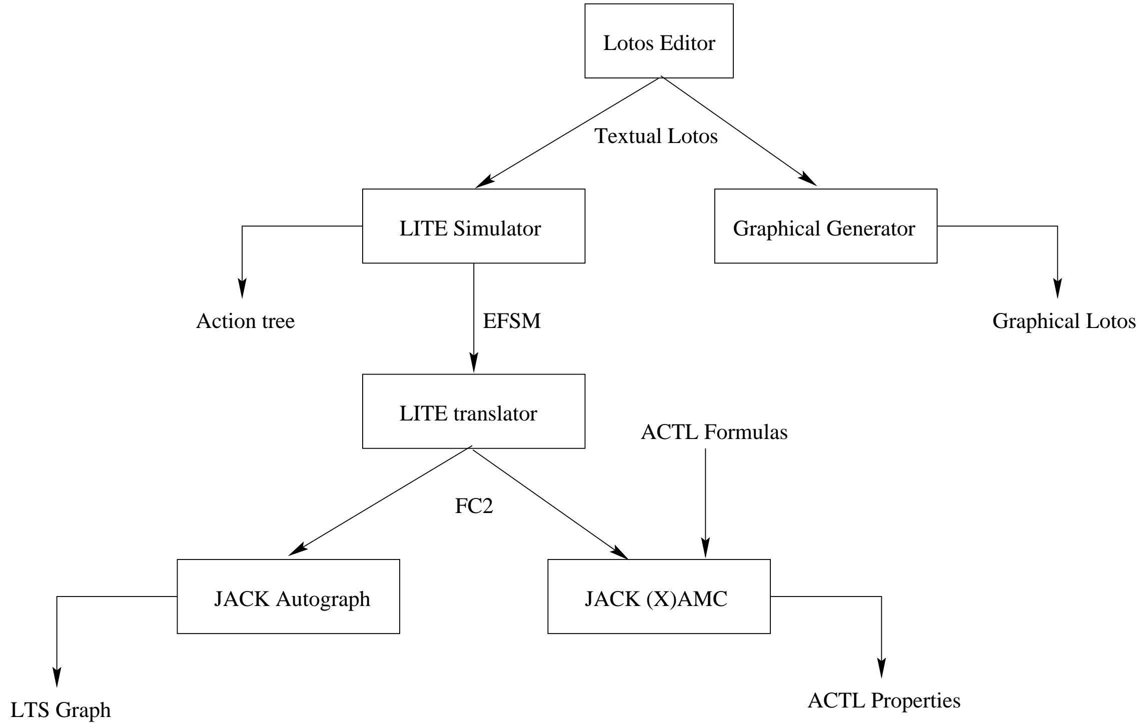


Figure 2: Specification and Verification Tools

The textual basic Lotos specification can be presented in different ways. The full Lite environment provides both tools for graphical representation and tools for simulation and for the generation of Extended Finite State Machines. The simulator can be used to derive an action tree (where each path in the tree represents one of the traces that can possibly be generated) and to produce an Extended Finite State Machine (EFSM) which is strong bisimulation equivalent to the original behaviour expression. Extended Finite State Machines can be translated automatically into an automaton in a format (FC2) suitable for input to the JACK toolset. This set comprises (M)AUTO, Autograph and the (X)AMC model checker. In particular, we can generate graphical representations of the automata with the Autograph tool such as those presented in figures 3 and 4. More important, we can use the (X)AMC model checker to automatically verify properties of a specification that are written as ACTL formulas. ACTL is an action based temporal logic. Its syntax and informal semantics are summarized in the annex. Whenever a formula does not hold, the model checker can produce traces of actions that show the violation of the formula. This provides useful information on how to improve the specification. Figure 2 outlines the various transformations of the Lotos specification and the verification tools we used to analyse and develop the specification discussed in this paper.

4 The Behaviour of Synchronizable Objects

4.1 Basic Transitions of Modes

The synchronizable object type defines operations for making transitions between four different synchronization modes: STOPPED, PLAYING, PAUSING and WAITING. The initial mode is STOPPED. For each

mode, operations are defined that result in a transition to a different mode of the synchronizable object.

We describe the behaviour with respect to mode transitions by means of one Lotos process for each of the values defining the mode space. The behaviour of each process is defined as a non-deterministic choice between behaviour expressions, each one implementing one of the operations that can be invoked on the object in the mode represented by this process. This approach allows for the development of the specification per value of each control variable. The different parts of the specification can be combined by using a constraint oriented style of specification in the sense that the parts are combined by means of a synchronization operator. We show this in more detail while developing the specification step by step.

Following the informal specification of the standard and the formal definition of the variables making up the state space given in Object-Z, we define the following Lotos actions that direct the mode transition of synchronizable objects, distinguished as observable external actions and non-observable internally performed ones:

external actions:

- doSTOP directs a transition to STOPPED mode from any mode,
- doPLAY directs a transition to PLAYING mode when in STOPPED mode,
- doPAUSE directs a transition to PAUSING mode from PLAYING or WAITING mode,
- doRESUME directs a transition to PLAYING mode from PAUSING mode or from WAITING mode,
- exc preserves the current state and represents the raise of an exception caused by the invocation of an operation not allowed in this mode

internal actions

- doWAIT enters the WAITING mode from PLAYING mode (it is an abstraction for the Wait flag set at a reference point during progression)
- donePlay enters the STOPPED mode from PLAYING mode on completion of a playing cycle.

With the above definitions, a process describing the behaviour of the synchronizable object is one specifying its basic mode transition mechanism:

```
process SynchronizableObject[doSTOP,doPLAY,doPAUSE,doRESUME,exc] : noexit :=
    hide doWAIT,donePlay in
    modeTransitions[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc
```

The basic mode transition mechanism itself is specified by a process that behaves as in STOPPED mode, which is the initial mode of each synchronizable object:

```
process modeTransitions[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc] : noexit :=
    STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc
```

When in STOPPED mode, only a play or a stop operation can be invoked, while any other request results in an exception. Consequently, the corresponding process offers the choice of performing a doPLAY or a doSTOP action, or to raise an exception. The creation of a PLAYING process models the fact that the process goes into PLAYING mode. If an exception is raised or a doSTOP is performed, the mode does not change, so the process remains in STOPPED mode.

```
process STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc] : noexit :=
    doPLAY; PLAYING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doSTOP; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    exc; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc
```

When playing, synchronizable objects may respond to external as well as to internal operation invocations. The current play process can be explicitly stopped and enter STOPPED mode or it can be paused and enter PAUSING mode. Any other invocation of externally observable actions results in an exception. However, a synchronizable object can perform mode transitions as a consequence of internal actions. When the playing cycle is completed, it enters the STOPPED mode and we model this by the donePlay action. Furthermore, it enters the WAITING mode when a reference point is encountered during progression with the Wait flag set. This behaviour is captured by the following process:

```
process PLAYING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc] : noexit :=
    doSTOP; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doPAUSE; PAUSING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    exc; PLAYING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    donePlay; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doWAIT; WAITING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc
```

Similar reasoning applies to the specification of the processes defining the behaviour of synchronizable objects when in PAUSING and WAITING mode. In the former, it is possible either to resume playing by invoking the doRESUME action, or to stop. In the latter, it is possible to choose between resuming the play process, stopping, or pausing. In both modes an exception is raised in the case an operation is invoked other than those specified. The corresponding Lotos processes are defined as in the following:

```
process PAUSING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc] : noexit :=
    doRESUME; PLAYING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doSTOP; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    exc; PAUSING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc

process WAITING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc] : noexit :=
    doRESUME; PLAYING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doSTOP; STOPPED[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    doPAUSE; PAUSING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    []
    exc; WAITING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
endproc
```

The modeTransitions process is a model of the basic transition mechanism of synchronizable objects. The corresponding automaton consists of four states, each one representing a mode, and fifteen transitions as shown in Figure 3.

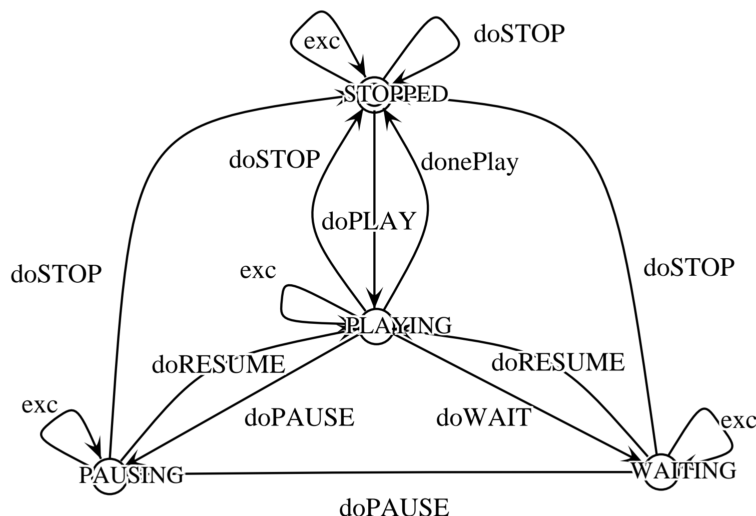


Figure 3: Mode transition diagram

4.2 Analysis of mode transition behaviour

The automata, generated as a result of simulating the modeTransistions process with the verification toolset of Lite, can be shown automatically to satisfy safety properties expressed as ACTL formulas. We list a number of interesting ones as an example. For reasons of lack of space we refer to [DFGR91] for an explanation of the details of the ACTL formulas and give in Appendix A an overview of its syntax and informal semantics. Here, we give an informal description of each property that has been verified. We are aware of the fact that the informal descriptions may not capture the property precisely enough but this shows at the same time why a formal description is helpful in avoiding ambiguities and misunderstandings.

In order to address internal actions in the temporal logic formulas we made these actions observable in the specification before model checking.

1. In all states it is always possible to raise an exception

   $$AG < exc > true;$$

2. Whenever a doRESUME is observed in a trace this must have been preceded by at least one doPAUSE or doWAIT since the last doRESUME or, if lacking, the beginning of the trace.

   $$\sim\ E\,[true\,\{\sim\ (doPAUSE \mid doWAIT)\}\ U\ \{doRESUME\}\ true]\&$$

   $$AG\,([doRESUME]\ \sim\ E\,[true\,\{\sim\ (doPAUSE \mid doWAIT)\}\ U\ \{doRESUME\}\ true])$$

3. Whenever a doPLAY occurs, then whenever a doPAUSE or doWAIT occurs, eventually a doRESUME is enabled.

   $$AG\,[doPLAY]\ AG\,[doPAUSE \mid doWAIT]\ AF\ < doRESUME >\ true$$

   The following property is *not* satisfied:

4. Whenever a doPAUSE or doWAIT occurs in a trace this must have been preceded by a doPLAY since the last doPAUSE or doWAIT or, if lacking, the beginning of the trace.

$$\sim\ E\,[true\,\{\sim\ doPLAY\}\ U\ \{doPAUSE \mid doWAIT\}\ true]\&$$

$$AG\,([doPAUSE \mid doWAIT]\ \sim\ E\,[true\,\{\sim\ doPLAY\}\ U\ \{doPAUSE \mid doWAIT\}\ true])$$

But this is also good because indeed a doPAUSE can be done directly after a doWAIT without doPLAY in between! The model checker is able to give feedback on the results by presenting traces of actions that show the violation of a property; in this case:

$$\rightarrow doPLAY \rightarrow doWAIT \rightarrow doPAUSE$$

5 Specification Refinement

Having described the basic transitions of modes, the next step is to get more insights into the system behaviour by introducing more details to refine the previous specification. An interesting refinement is the specification of the meaning of progression through the co-ordinate space when the object is playing. To extend the specification two approaches are possible. Since the refinement is addressing operations that occur in PLAYING mode, a natural choice would be to redefine the behaviour of the PLAYING process by adding further expressions that take into account the progression. Alternatively, a new process can be specified describing only the behaviour of progressing the position. This process is subsequently composed with the modeTransitions process following a constraint oriented style of specification. This second alternative presents a number of advantages including modularity and incremental refinement of the specification by cumulating locally defined constraints. Consequently, we adopt a constraint oriented style of specification and define a new process to model progression.

When directed to start a playing cycle, synchronizable objects enter a loop in which the next required position is computed progress towards that position by performing a sequence of steps. Such a sequence is called a stage. While stepping, the signaling of an event occurs in case a reference point is encountered. Stepping is terminated when the object reaches the required position or because it completes playing. New actions are defined to model the above behaviour:

- external actions:
  - doSignal models the signaling of a reference point;
- internal actions:
  - target models the computation of the next required position;
  - doStep models the progression of the position;
  - doneStage models the termination of progression because the target has been reached.

Because each stage is to be considered atomic, the progression process has to delay external operations until an appropriate moment, which is at the end of a stage or, for a limited number of external operations, when in WAITING state. Consequently the external operations appear as gates in the process definition.

The process describing the progression of the position initially creates an instance of a process offering the choice of possible behaviour when the object is not stepping.

```
process progressPosition[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,doSignal] :
noexit :=
    hide doStep,doneStage in
    NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                doSignal,doStep,doneStage]
endproc
```

The stepping stage is performed only when the synchronizable object is in PLAYING mode. Entering PLAYING mode is characterized by the occurrence of either a doPLAY or a doRESUME. Consequently the NOTSTEPPING process reacts to a doPLAY or a doRESUME by computing a new required position with the NEWTARGET process which leads to the stepping stage, or performs a simple recursion in the case of doSTOP or doPAUSE. Eventually, the refinement of the doStep and doneStage actions will require the reworking of this process definition to take into account the special needs of different media types.

```
process NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                    doSignal,doStep,doneStage] :
noexit :=
    doPLAY; NEWTARGET[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                      doSignal,doStep,doneStage]
    []
    doRESUME; NEWTARGET[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                        doSignal,doStep,doneStage]
    []
    doSTOP; NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                        doSignal,doStep,doneStage]
    []
    doPAUSE; NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                         doSignal,doStep,doneStage]
endproc
```

The computation of a new required position is modelled by the target action in the NEWTARGET process. A more concrete specification will require the reworking of this process to match the characteristics of specific media types. After the target has been computed the STEPPING stage is entered.

```
process NEWTARGET[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                  doSignal,doStep,doneStage] :
noexit :=
    hide target in
    target; STEPPING [doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                      doSignal,doStep,doneStage]
endproc
```

The STEPPING process defines at a high level of abstraction the possible actions that may take place during one stage progression. The behaviour initially offered in a choice is either to perform a progression step, or to notify the completion of a stage or of a play cycle. When a stage is completed a new one can be entered by computing a new target, or the progression can be terminated. The completion of a play cycle may always occur and modelled by the donePlay action.

In the specification of the STEPPING and NOTSTEPPING processes we have been using non-determinism to abstract from the Object-Z specification. Instead of modelling all the variables that describe the precise movement of a pointer in the coordinate space, we modelled only the events that trigger a change in the behaviour, such as the fact that the pointer reaches the end position of a playing cycle that generates a donePlay. In the specification we abstract from *when* exactly such events occur and model only *that* such an event can occur by means of a non-deterministic choice of the possible events. This way we capture everything that *may* happen and we can investigate all possibilities.

When the object is progressing its position by stepping, it can encounter a reference point. In this case it signals the event and may enter the WAITING mode if the Wait flag is set. According to the specification of mode transition, after the WAITING mode is entered the object can be either stopped or paused or resumed. It should be noted however that this is not equivalent to re-instantiate the NOTSTEPPING process. In fact, the doRESUME action continues the playing cycle with exactly the same stage that was active before

waiting, when performed in the WAITING mode. Conversely, the same action initializes a new stage by computing a new target when performed in the PAUSING mode.

```
process STEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                 doSignal,doStep,doneStage] :
noexit :=
    doStep; (
       STEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                doSignal,doStep,doneStage]
       []
       doSignal; (
          STEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                   doSignal,doStep,doneStage]
          []
          doWAIT; (
             doRESUME;
                STEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,
                         donePlay,doSignal,doStep,doneStage]
             []
             doSTOP;
                NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,
                            donePlay,doSignal,doStep,doneStage]
             []
             doPAUSE;
                NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,
                            donePlay,doSignal,doStep,doneStage] ) ) )
    []
    doneStage; (
       NEWTARGET[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                 doSignal,doStep,doneStage]
       []
       NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                   doSignal,doStep,doneStage] )
    []
    donePlay; NOTSTEPPING[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                          doSignal,doStep,doneStage]
endproc
```

By combining the definition of modeTransitions and progressPosition processes, the refined version of synchronizable objects is defined as:

```
process SynchronizableObject[doSTOP,doPLAY,doPAUSE,doRESUME,exc,doSignal] :
noexit :=
    hide doWAIT,donePlay in
    modeTransitions[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    |[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay]|
    progressPosition[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,
                     doSignal]
endproc
```

The two processes are composed in parallel with synchronization on the gates doSTOP, doPLAY, doPAUSE, doRESUME, doWAIT and donePlay. Consequently, the processes constrain each other so that

progressPosition cannot force a mode transition that would violate the transition mechanism specified by modeTransitions and, conversely, modeTransitions let the object make a mode transition only at those points where also progressPosition is enabled to perform the same action.

The simulation of the specification shows, as expected, a more complicated transition system shown in Figure 4 where the states in the middle that are surrounded by two boxes indicate the refinement of the PLAY mode with respect to Figure 3. The small box contains the states representing NOTSTEPPING, the large box those representing STEPPING.

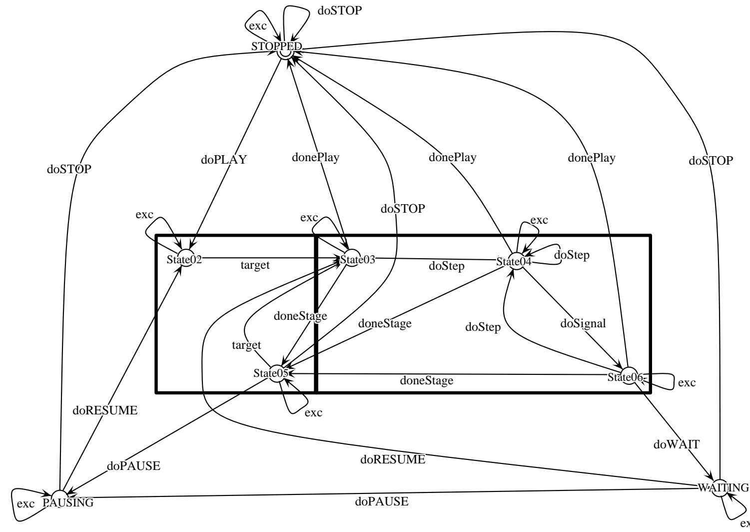


Figure 4: Synchronizable object transition diagram; modes and progression

5.1 Analysis of mode transition and progression

A new set of properties hold in the refined version of the automata. In particular, the properties expressed for the mode transition part in Section 4.2 hold also for the refined specification. In addition the following properties hold that are specific for the extended specification and that have also been formulated in the Foundation Component document of the PREMO standard [ISO96] on section "7.9 Synchronization".

1. After a doRESUME from the PAUSING state always a new target is computed before stepping is continued.

   $$AG\ [doPAUSE][doRESUME]\ \sim\ E[true\ \{\sim target\}\ U\ \{doStep \mid donePlay \mid doneStage\}\ true]$$

   Note that continuation of stepping is indicated by the occurrence of doStep, donePlay or doneStage.

2. After a doRESUME from the WAITING state it is *not* the case that first a new target is computed before stepping is continued.

   $$AG[doWAIT][doRESUME] \sim E[true\{\sim (donePlay \mid doStep \mid doneStage)\}U\{target\}true]$$

6 Adding a Setting Operation: Jump

So far we have been modelling the basic operations of the Synchronizable Object. There are two more kinds of operations to be added to this object; information retrieve actions and value setting actions. Retrieve

actions are those that can be invoked to get information on for example the current mode of the object. These actions do not change the state of the object and we don't expect interesting behavioural problems. The setting actions, on the other hand, may change the state of the object and since the behaviour is depending on these values it is worth to perform a careful analysis. In this paper we investigate the consequences of adding a jump-operator that can change the value of the "currentPosition" pointer to illustrate the kind of analysis that can be performed based on the specification given earlier.

The essential design question we are interested in here is under which conditions it is safe to allow a jump-action to be invoked on the synchronizable object. To be able to formulate a proper safety condition for this situation we have to take a closer look at the progress position algorithm.

When a synchronizable object enters the playing mode, it starts a sequence of one or more playing stages. A stage consists of several small steps through the coordinate space where the position is recorded in a separate temporary pointer called "pointer" in Figure 1. The end of each stage is marked by the "requiredPosition" (modelled by the target action) which is computed at the beginning of each stage. The "currentPosition" pointer is only updated after a target has been reached. A stage can be interrupted if a "reference point" is encountered in which the synchronizable object is required to wait and go into WAITING mode. A possibly dangerous situation can occur when the "currentPosition" pointer is changed after a new required position has been computed and before the currentPosition pointer has been updated to a stable value, either by reaching the target, or by encountering the end of the coordinate space (modelled by "donePlay") or by a doSTOP or a doPAUSE action which both may occur when the synchronizable object enters WAITING mode.

In terms of the model checker we should verify in every design alternative that it never happens that a jump occurs between the calculation of a new target and the occurrence of one of the actions doneStage, donePlay, doSTOP and doPAUSE. In ACTL we have to formulate this as follows. In every path at every node that can be reached by performing "target" there does not exist a path in which we cannot observe one of the actions doneStage, donePlay, doSTOP or doPAUSE, which characterize the end of a stepping stage, before we can observe a jump. Formally:

$$AG[target](\sim\ E[true\{\sim\ (doneStage|donePlay|doSTOP|doPAUSE)\}U\{jump\}true])$$

In the following four sections we investigate different design variants. In each variant the jump operation is allowed under different conditions. These conditions are specified by adding a further constraint to the behaviour of the synchronizable object. We investigate the following four options:

- Jump always enabled
- Jump disabled only when in PLAYING mode
- Jump disabled when in PLAYING or WAITING mode
- Jump disabled only when STEPPING during a stage

For all the variants we use the same structure for restricting the situations in which the jump operation is enabled. We add a process called "setOp" (setting operations) to the main behaviour of the synchronizable object in a constraint oriented way. The behaviour is specified as follows:

```
process SynchronizableObject[doSTOP,doPLAY,doPAUSE,doRESUME,exc,doSignal,jump] : noexit :=
    hide doWAIT,donePlay in
    modeTransitions[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,exc]
    |[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay]|
    progressPosition[doSTOP,doPLAY,doPAUSE,doRESUME,doWAIT,donePlay,doSignal]
    |[...]|
    setOp[jump,...]
where (* remaining process definitions *)
```

The process definitions of modeTransitions and progressPosition are unchanged. The jump operation is introduced only by means of the setOp process. The set of gates on which setOp synchronizes with progressPosition and modeTransitions vary depending on the definition of setOp.

6.1 Jump always enabled

In this section we investigate the possibility to enable jump in all situations. In this case the process setOp is very simple and is put in interleaving with the processes modeTransitions and progressPosition.

```
process setOp[jump] : noexit :=
    JUMPING[jump]
where
    process JUMPING[jump] : noexit :=
        jump; JUMPING[jump]
    endproc
endproc
```

As we could expect, the model checker gives as a result that the formula under investigation, formally expressed in the previous section, is FALSE for this specification. But, moreover, the model checker can be asked to give an example of a trace that violates the formula. In this case it shows that the trace

$$\rightarrow doPLAY \rightarrow target \rightarrow jump$$

is an example of a trace that violates the property we have formulated.

6.2 Jump disabled only during PLAYING mode

Due to the result shown in the previous section, we investigate if it is enough to disable jump during PLAYING mode. The process setOp can observe if the synchronizable object enters and leaves PLAYING. The entering of PLAYING mode is triggered by an occurrence of doPLAY or doRESUME. This is modelled in the process setOp that enables jumping unless a doPLAY or a doRESUME action is observed. In that case jump is disabled which is modelled by the JmpDisable process.

```
process setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT] :
noexit :=
    JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
    [>
    ( doPLAY; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
      []
      doRESUME; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT] )
endproc
```

When jump is disabled the process JmpDisable makes sure that jump is indeed not available. It also checks for the occurrence of the actions doPAUSE, doSTOP, doWAIT and donePlay that indicate that the synchronizable object is leaving the PLAYING mode and thus trigger the fact that jump has to be enabled again. It also enables the doRESUME action so that it can take place, but since the occurrence of doRESUME does not mean that jump can be enabled it remains disabled by continuing with JmpDisabled.

```
process JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT] :
noexit :=
    doPAUSE; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
    []
    doSTOP; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
```

```
    []
    donePlay;setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
    []
    doWAIT; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
    []
    doRESUME; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME,doWAIT]
endproc
```

Since the process setOp is synchronized on several actions, the JUMPING process has to guarantee that those actions can always occur also when jumping is enabled. Therefore JUMPING is modelled as the choice of jump and all actions that appear in the synchronization and that do not indicate that the synchronizable object enters PLAYING mode.

```
process JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT] :
noexit :=
    jump; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
    []
    doPAUSE; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
    []
    doSTOP; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
    []
    donePlay; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
    []
    doWAIT; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay,doWAIT]
endproc
```

The process setOp is synchronizing on the actions doPLAY, doPAUSE, doSTOP, donePlay, doRESUME and doWAIT with the processes modeTransitions and progressPosition.

Also in this case the model checker shows that the formula does *not* hold. So disabling jump only during playing is clearly not enough. The trace given by the model checker that violates the requirement is

$$\rightarrow doPLAY \rightarrow target \rightarrow doStep \rightarrow doSignal \rightarrow doWAIT \rightarrow jump$$

6.3 Jump disabled during PLAYING and WAITING

In order to disable jump when the synchronizable object is in PLAYING or WAITING mode the enabling of jump has to be restricted. The structure of the process setOp is similar to the one of the previous case except that we don't need to take the doWAIT action explicitly into consideration. In fact, the only way to enter WAITING mode is from PLAYING and the same set of actions identifies the leaving from both modes.

```
process setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME] :
noexit :=
    JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay]
    [>
    ( doPLAY; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME]
      []
      doRESUME; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME] )
endproc
```

The process JmpDisable makes sure that jump is indeed not available and enables it again only after the occurrence of doPAUSE, doSTOP or donePlay. As in the previous case, it also enables the doRESUME action so that it can take place.

```
process JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME] :
noexit :=
    doPAUSE; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME]
    []
    doSTOP; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME]
    []
    donePlay; setOp[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME]
    []
    doRESUME; JmpDisable[jump,doPLAY,doPAUSE,doSTOP,donePlay,doRESUME]
endproc
```

Like in the previous example the process JUMPING enables jump and guarantees that other actions on which setOp synchronizes do not get blocked.

```
process JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay] :
noexit :=
    jump; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay]
    []
    doPAUSE; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay]
    []
    doSTOP; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay]
    []
    donePlay; JUMPING[jump,doPLAY,doPAUSE,doSTOP,donePlay]
endproc
```

The process setOp is synchronizing on the actions doPLAY, doPAUSE, doSTOP, donePlay and doRESUME with the processes modeTransitions and progressPosition.

After this we can check that the formula under investigation is TRUE for this specification. So with this result we can conclude that disabling jump when the synchronizable object is in PLAYING or WAITING is a safe solution with respect to the property we are investigating.

6.4 Jump disabled only during STEPPING

There exists however a less restrictive alternative than the previous one that still satisfies the required property. This is the option to disable jump only during the situation in which the synchronizable object is in stepping mode. This means that there may be a number of moments during that situation in which a jump action can safely be allowed.

To show this we follow again a similar structure as in the previous examples. Entering stepping mode in this case is triggered by the action "target" that models the calculation of a new required position to be reached in the current stage.

The stepping mode is left when one of the actions doPAUSE, doSTOP, doneStage or donePlay are encountered. The process setOp synchronizes with the modeTransitions and progressPosition processes on the actions doPAUSE, doSTOP, donePlay, target and doneStage.

```
process setOp[jump,doPAUSE,doSTOP,donePlay,target,doneStage] :
noexit :=
    JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
    [>
    ( target; (
        doPAUSE; setOp[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        doSTOP; setOp[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
```

```
        []
        doneStage; setOp[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        donePlay; setOp[jump,doPAUSE,doSTOP,donePlay,target,doneStage] ) )
where
    process JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage] :
    noexit :=
        jump; JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        doPAUSE; JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        doSTOP; JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        donePlay; JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
        []
        doneStage; JUMPING[jump,doPAUSE,doSTOP,donePlay,target,doneStage]
    endproc
endproc
```

The model checker shows that indeed for this design variant the safety property holds.

It should be noted that this specification requires a small reworking of progressPosition. Since setOp relies on a synchronization occurring at gates target and doneStage, they must be explicitly shown in the list of gates of progressPosition and be hidden within SynchronizableObject.

6.5 Evaluation of the four variants

There are a number of interesting observations that can be made based on the four variants we analysed in the previous sections.

First of all it is shown that with a constraint oriented approach there is the possibility to extend the specification with other operations in a modular way, i.e. without changing the processes that had already been defined. This has been made possible due to a number of methodological decisions. The most important is the choice for a constraint oriented approach and further the particular way this has been worked out in the specification. The essential control variables in the Object-Z specification have been defined as processes that reflect the different values these control variables can take and the process definitions clearly indicate which operations cause a change in their value.

The constraint oriented approach made it possible to formulate setOp as a separate constraint without changing the other processes. The clear structure in the effect of actions on the value of the control variables made it possible to formulate the different constraints in an explicit way.

Of course it is not guaranteed that all possible design variants can be investigated with a minimal effort in this way. But at least many options can be evaluated as long as no essential changes of the structure of the main specification are required. For example this may be the case if a jump operation *must* be possible during stepping at any time without delay. This would make jump an operation that has influence on the value of the control variables that keep track of the stepping cycle of the synchronizable object, and thus requires a change in the process progressPosition.

The specification of the setOp processes indicates clearly which actions are considered to trigger the entering in or leaving from particular situations. These may be very helpful in checking the Object-Z specification because these actions essentially indicate at which points certain variables *have* to be updated. In these variants this has been made explicit for the pointer that indicates the current position in the coordinate space of the synchronizable object.

7 Conclusion

In this article we illustrated a way to obtain a Basic Lotos specification from the Object-Z specification of the PREMO synchronizable object. We have shown how such a specification can be obtained by motivating rigorously, although not fully formal, the relation between the two specifications. The essential idea is to identify the control variables in the Object-Z specification and to model each value of each control variable as a separate process. The processes belonging to one control variable have been grouped in one process that consequently has been composed in parallel with other processes modelling other control variables. In fact this way we obtained a specification in a constraint oriented style. This way we also circumvented the problem of obtaining a full Lotos specification that is not directly suitable for model checking and whose automatic transformation into Basic Lotos may lead to a specification that is not equivalent to the full Lotos one.

The Basic Lotos specification serves as a formal model of the *behaviour* of the synhronizable object that can be analysed in detail by means of the tools that have been developed for Basic Lotos and for automata such as an action model checker. This way properties of the behaviour, formalized in the action temporal logic ACTL, could be automatically checked on the behavioural specification and the results directly confronted with the Object-Z specification. In this paper we illustrated that with relatively little effort different design options can be formally evaluated and important properties be checked using tools for formal methods. This provides a better founded evaluation of design options because of the availability of formally validated correctness results that can be taken into consideration in a more complete evaluation. This illustrates well that formal methods do not only play a role in the verification of properties after the specification is completed, but allow for a specification approach in which specification and verification are much more interleaved.

The integration of different tools are in this respect certainly a step forward because it allows one to work basically with one specification of which different aspects can be investigated. The different ways to present a specification may also play an important role in discussions on the design in which people with different expertise take part.

We also want to point out the interesting interplay between the two formal specification approaches. The Object-Z specification leads to a clear definition of relatively small objects whose behaviour can subsequently be modelled and investigated in an independent way by using other specification techniques and tools.

Of course, the more specific the questions are that one wants to investigate, the more detailed the model has to be, and this may lead to a trade-off between effort put into modelling and other ways to investigate the specification. Also in our method the level of abstraction was not directly derived from the Object-Z specification or the English standard. This part of modelling remains a matter of ingenuity of the designers which is probably inherent to the task of modelling.

8 Acknowledgements

This work has been carried out within the ERCIM Computer Graphics Network - Contract No. CHRX-CT92-0085, the Interactionally Rich Systems Network - Contract No. CHRX-CT93-0099 and as part of the Community Training Project Interactionally Rich Immersive Systems - Contract No. CHBG-CT94-0674 funded by the European Union under the Human Capital and Mobility Programme.

References

[BFe96] C. Bernardeschi and G. Ferro. A Sample Study Exemplifying the Use of JACK. In: Proceedings Workshop on Automated Formal Methods, ENTCS, vol. 5, University of Oxford, 1996.

[Bic94] J.C. Bicarregui et al. Proof in VDM: A Practitioner's Guide. Springer-Verlag, 1994.

[BGL94] A. Bouali and S. Gnesi and S. Larosa. The Integration Project for the JACK Environment. Bulletin of the EATCS, 54, October 1994, pp. 207-223.

[DDHF97] D. J. Duke and D. A. Duce and I. Herman and G. Faconti. *Specifying the PREMO Synchronization Objects* ERCIM Computer Graphics Network Technical Report, 02/97-R048, February, 1997. (submitted for publication)

[DFGR91] R. De Nicola and A. Fantechi and S. Gnesi and G. Ristori. *An action based framework for verifying logical and behavioural properties of concurrent systems.* In: Proceedings 3rd International Workshop CAV'91, K. G. Larsen and A. Skou (Eds.), LNCS 575, Springer-Verlag, 1991

[DFGR95] R. De Nicola and A. Fantechi and S. Gnesi and G. Ristori. Verifying Hardware Components within JACK. In: Proceedings of CHARME'95, LNCS 987, Springer-Verlag, 1995, pp. 246-260.

[DNNRA93] R.B. Dannemberg and T. Neuendorffer and J. Newcomer and D. Rubine and D. Anderson *Tactus: Toolkit-level support for synchronized interactive multimedia* Multimedia System Journal, 1(2):77–86, 1993.

[DRS94] R. Duke and G. Rose and G. Smith. *Object-Z: A Specification Language Advocated for the Description of Standards* Technical Report, No. 94-45, Software Verification Research Center, Department of Computer Science, University of Queensland, 1994.

[FM97] G. P. Faconti and M. Massink. *PREMO Synchronizable Objects: Specification and Verification* January, 1997. (submitted for publication)

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of ACM*, 21(8), 1978.

[ISO89] International Standards Organization ISO. Information processing systems, open systems interconnection, LOTOS. A formal description technique based on the temporal ordering of observational behaviour. ISO 8807, 1989.

[ISO96] International Standards Organization ISO. Information processing systems, computer graphics, presentation environment for multimedia objects (PREMO). ISO/IEC 14478, 1996.

[MaV90] E. Madeleine and A. Pnueli. *AUTO: A Verification tool for Distributed Systems using reduction of Finite Automata Networks.* Formal Description Techniques, II (S. T. Vuong, ed.), 1990, pp. 61-66.

[Mil89] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice Hall, 1989.

[RDD94] G.J. Reynolds and D.A. Duce and D.J. Duke Report of the ISO/IEC JTC1/SC24 special rapporteur group on formal description techniques. Technical Report ISO/IEC JTC1/SC24 N1152, ISO, 1994.

[RDS90] V. Roy and R. De Simone. *AUTO and Autograph.* In Proceedings Workshop on Computer Aided Verification, LNCS 531, Springer-Verlag, 1990, pp. 65-75.

[Spi89] J.M. Spivey. The $\mathcal{Z}$ Notation: A Reference Manual. Prentice Hall, 1989.

[vEi91] P. van Eijk. The Lotosphere Integrated Tool Environment LITE. In *Proceedings of 4th International Conference on Formal Description Techniques.* North Holland, 1991.

[VSv91] C. A. Vissers and G. Scollo and M .van Sinderen and E. Brinksma. *Specification Styles in Distributed Systems Design and Verification.* Theoretical Computer Science, vol. 89, 1991.

A Syntax and Informal Semantics of ACTL

In the following table the syntax and informal semantics are listed of the ACTL operators we used in this paper to formulate the temporal logic properties that have been checked by means of the (X)AMC model checker. For further explanation and a formal semantics we refer to [DFGR91].

Action formulas			
χ	::=	*true*	"any observable action"
		false	"no observable action"
		a	"the observable action a"
		$\sim \chi$	"any observable action different from χ"
		$\chi \mid \chi'$	"either χ or χ' "
		$\chi \;\&\; \chi'$	"both χ and χ'"
χ'	::=	χ	
State formulas			
ϕ	::=	*true*	"any behaviour is possible"
		false	"no behaviour is possible"
		$\sim \phi$	"ϕ is impossible"
		$\phi \;\&\; \phi'$	"ϕ and ϕ'"
		$E\gamma$	"there exists a possible execution in which γ"
		$A\gamma$	"for each of the the possible executions γ"
		$<\ \chi\ >\phi$	"there exists a next state reachable by an action that satisfies χ, in which ϕ holds "
		$[\chi]\phi$	"for all next states reachable with actions that satisfy χ, ϕ holds"
ϕ'	::=	ϕ	
Path formulas			
γ	::=	$[\phi\{\chi\}U\{\chi'\}\phi']$	"actions that satisfy χ are performed and ϕ holds *until* an action that satisfies χ' has been performed and then ϕ' holds"
		$[\phi\{\chi\}U\phi']$	"actions that satisfy χ are performed and ϕ holds *until* ϕ' holds"
		$G\phi$	"in every future state ϕ holds"
		$F\phi$	"there exists a future state in which ϕ holds"
Satisfaction relation for action formulas			
$a \models b$	iff	$a = b$	(a and b actions)
$a \models \sim \chi$	iff	$a \not\models \chi$	χ action formula
$a \models \chi \mid \chi'$	iff	$a \models \chi$ or $a \models \chi'$	
$a \models \chi \;\&\; \chi'$	iff	$a \models \chi$ and $a \models \chi'$	