Synchronizable Objects in Distributed Multimedia Applications

Jun Hu and Loe Feijs

Department of Industrial Design
Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
{j.hu,l.m.g.feijs}@tue.nl

Abstract. In training and gaming systems, distributed multimedia are often used, in which the basic content elements must be conveyed or presented in a synchronized order at synchronized moments over multiple devices and in many cases over a network. These content elements are often presented or represented as "Synchronizable Objects" with which their control and management fall into a design pattern. This paper uses the pattern language to capture the common features of these "Synchronizable Objects", in combination of the formal Object-Z specification to treat the architectural construct. The proposed pattern can be applied for content elements with or without intrinsic timing in distributed multimedia applications. Examples are given to show how this pattern can be applied in distributed applications.

Keywords: multimedia, synchronizable object, design pattern, formal specification.

1 Introduction

Training and gaming systems often use distributed multimedia (text, audio, graphics, animation, video, and recently movements and behaviors of physical objects), to convey information in synchronized order and at synchronized moments over multiple devices and in many cases over a network. These content elements are often presented or represented as "Synchronizable Objects" with which their control and management fall into a design pattern. In this article we try to follow the pattern language [1] to describe its intent, context, forces, solution and consequences. Often in object-oriented design patterns, the solution is described using the Unified Modeling Language (UML) in combination of examples of source code. However UML as such is not sufficient to capture the details of the common features in pattern specification, and examples of source code are often full of too much of implementation details [2]. A formal and object-oriented specification language, Object-Z [3,4], is then used to compensate the insufficiency of UML. We then give two examples in which the design pattern is applied, one in gaming, the other in training.

2 Synchronizable Object

2.1 Intent

The Synchronizable Object (SO) pattern extracts the common inter-media synchronization behavior from the media objects with intrinsic timing and the ones without, such that distributed media objects can be synchronized at distinct and possibly user definable synchronization points.

2.2 Context and Forces

Many media objects have intrinsic timing (audio, video, animation), and some don't but require the content to be presented in a given order (for example, speech [2,5,6] and movements [7]), where as the others are static (text, image and graphics). Some media appear active (video, audio, animation and TTS) and have a automatic and successive behavior, whereas some others are passive (presentation slides, linked web pages) and require external drive to move forward. How to synchronize all these different media is not trivial. Especially in an ambient intelligent environment, media are considered in a generic sense, including both digital and physical objects. Designers have the freedom to create new forms of media hence new media types should be able to be incorporated into the system easily.

In JMF[8], timed media objects or streaming media objects that have intrinsic timing, for example, audio and video objects, are treated differently from media objects that do not have intrinsic timing. While synchronization mechanisms (Libraries and APIs) are well defined for timed media objects, other media objects such as graphics and text are treated as different data types for which no synchronization mechanisms are defined. To synchronize with the timed media objects, the controlling interfaces of streaming media have to be implemented on top of these non-timed types, which is not only unnatural, but also sometimes hard to impose the semantics of timed media on the non-timed ones.

In PREMO [9] and STORYML[10,11,12], all the media objects are assumed to be active, that is, every object has its own process driven by a *Timer* (PREMO) or a *MediaClock* (STORYML), even if the object does not have intrinsic timing. This approach simplifies the synchronization by providing a common time-based mechanism for all the objects, however may increase the process resource overload caused by the unnecessary active process that need not to be actually active.

Moreover, when distribute these media objects over a network, it is necessary to clearly sperate the timing concerns and the synchronization control and to minimize the communication load [2,13].

2.3 Solution

The design of this pattern is very much inspired by the structure appeared in many multimedia standards and systems, especially JMF, PREMO amd IPML [2]. The formal specification of the synchronization objects is based on

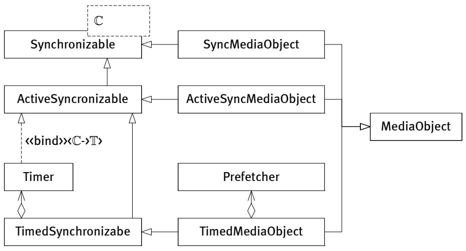

Fig. 1. Synchronizable Objects, where the *Synchronizable* can be distributed

the foundational objects described in [14], omitting exception handling mechanisms that are considered not to be important in a pattern specification, and applying a different event handling design. The design is then extended with the notion of time and the controlling structure inspired by JMF.

Structure. As the basis, the concept of the coordinate system is introduced. Every SO has a progressive behavior – to be started, step forward until the end if there is an end, and possibly repeat from the beginning. A step is modeled as a coordinate point and the sequence of the steps forms a coordinate system. An *ActiveSynchronizable* object automatically steps forward in its own process and a *TimedSynchronizable* automatically moves forward but also with timing constraints. The timing constraints are applied to the *TimedSynchronizable* by synchronizing with a *Timer*, which itself is an *ActiveSynchronizable* with a moving-forward or a moving-backward timeline at a specific speed. Synchronizable media objects are SO's at all the levels of the inheritance hierarchy, but also implement the presentation behavior as a *MediaObject*. In addition, a *TimedMediaObjects* needs a *Prefetcher* that keeps fetching enough amount of data ahead of the presentation process, so that immediate starting and continuous presentation are possible(Fig. 1).

The synchronization between SO's is event driven. A *Synchronizable* may trigger *StateSyncEvent*s when it changes its synchronization state (stopped, ready, started and paused), may also issue other events that are attached to a coordinate when the coordinate has been passed. Every SO dispatches the synchronization events to the event handlers that are interested in certain sets of events and sources. They an object of the type *EventDispatcher* that selects the registered *EventHandler* objects and dispatches the synchronization events of the interests. The *EventHandler* object in turn invokes concrete event handling operations in other objects. The *TimedSynchronizable* object synchronizes the internal state with a timer (reacting on the *StateSyncEvent*s), and present the data on the paces of that timer (reacting on the *TimedSyncEvent*s), which also demonstrates the use of this event-driven approach (fig. 2).

Synchronization states. A SO is a simple finite state machine with four states stopped, ready, started, paused. The initial state is stopped. A SO must get ready before it can be started. A ready state is necessary to model many

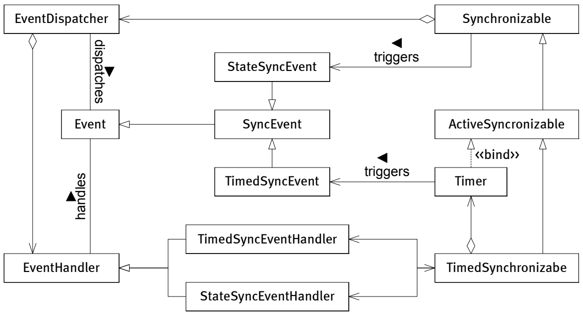

Fig. 2. Event based synchronization, in which the events can be be distributed

multimedia objects that requires computational resources to be allocated and certain amount of data to be prefetched before it can be started immediately. Once it has been started, it can be paused and be restarted again. A free type *SyncState* is introduced to identify these states:

$$SyncState ::= \mathsf{stopped} \mid \mathsf{ready} \mid \mathsf{started} \mid \mathsf{paused}$$

Coordinates. A SO controls the position and progress along its own coordinates. Different SO's may use different coordinates. For example, a video stream might use frame numbers as coordinates, and a clock might simply take $\mathbb{T}$ as its coordinates. A SO may have a function to map the coordinates to time and time constraints can be added to the operations – however the coordinates are not necessarily always time related, for example, a TTS engine may take white spaces or punctuation locations between words or sentences as coordinates. The concept of coordinates is more general than the time. To model the progression steps, the coordinates synchronization coordinates $\mathbb{C}$ is introduced and defined as a discrete set of discrete *points* that can be ordered with a relation $<$ that is irreflective, transitive, antisymmetric and total.

Repeating positions. A client of a SO might want to traverse the coordinates more than once, or possibly traverse in an infinite loop, for example, repeating a multimedia presentation several times or repeating it forever. This means a given point within the coordinates may be visited multiple times. A new type is defined to combine a point in the coordinates with a visit number, $Position == \mathbb{C} \times \mathbb{N}$, and a total order is defined over *Position*:

$$\_\ prec\ \_ : Position \leftrightarrow Position$$

$$\forall c_1, c_2;\ n_1, n_2 \bullet (c_1, n_1)\ prec\ (c_2, n_2) \Leftrightarrow n_1 < n_2 \lor (n_1 = n_2 \land c_1 < c_2)$$

Progression directions. A SO may advance along its coordinates, or backward in an inverted direction:

$$Direction ::= \mathsf{forward} \mid \mathsf{backward}$$

Synchronization Events. The coordinates can be attached with synchronization events. *Event* is defined as the super class for all the events in the system:

Event

$source : \mathbb{O}_{\circledcirc}$

$SetSource \mathrel{\widehat=} [\, \Delta(source)\ s? : \mathbb{O}_{\circledcirc} \mid source' = s? \,]$

which has an attribute *source* storing the reference to the originating object if desired[1]. Other details are left to subclasses. To distinguish synchronization events from other events (for example, user input events), *SyncEvent* is defined as a subclass of *Event*. Subtyping can be used to identify different types of *SyncEvent* objects, and the subclasses may also add other state variables to carry extra information. Instead of giving all the possible different types in advance, which is often not feasible, subtyping is a more flexible and extensible solution for identifying objects [15].

Synchronizable. Let's first give a visibility list that declares the visible state variables and operations:

Synchronizable

$\upharpoonright($ *eventDispatcher*, *repeatNumber*, *repeatCount*, *syncState*, *begin*, *end*, *current*, *syncElements*, *direction*, [Visible state variables]

INIT, *Start*, *TryStart*, *Stop*, *Pause*, *Visit*, *Step*, *setRepeatNumber*, *setDirection*, *setBegin*, *setEnd*, *setBeginEnd*, *attachSyncEvent*, *detachSyncEvent*, *addSyncElements*, *removeSyncElements* [Visible operations]

$)$

State variables. The state schema is then defined as follows:

Readable state variables:

$eventDispatcher : \downarrow EventDispatcher$

$repeatNumber : \mathbb{N}$ [How many times to repeat]

$repeatCount : \mathbb{N}$ [How many times repeating completed]

$syncState : SyncState$ [Current synchronization state]

$syncElements : \mathbb{C} \leftrightarrow \downarrow SyncEvent$ [Synchronization elements]

[1] The symbol $\mathbb{O}$ is defined as $\mathbb{O} == \downarrow Object$ where *Object* is the root superclass for all the classes in the system. $\downarrow Object$ is the set of all the objects of the class *Object* and its subclasses – the object universe of the system.

In this specification, a generic free type is also introduced to include the null value: $X_{\circledcirc} ::= X \mid \mathsf{null}$ for a given type X.

$begin : \mathbb{C}_{\circledcirc}$ [Where to begin]
$end : \mathbb{C}_{\circledcirc}$ [Where to end]
$current : \mathbb{C}_{\circledcirc}$ [Current location in the coordinates]
$syncSpan : \mathbb{P}\,\mathbb{C}$ [Synchronization span between *begin* and *end*]
$direction : Direction$ [traversal direction]

Local state variables that are invisible to others:
$positions : \mathbb{P}\, Position$ [Positions to be traversed]
$curPosition : Position_{\circledcirc}$ [Current position]
$lastPosition : Position_{\circledcirc}$ [The position visited last time]
$\_ \prec \_ : Position \leftrightarrow Position$ [Order of traversal]

$$syncSpan = (\textbf{if } begin = \mathsf{null}\ \textbf{then}\ \mathbb{C}\ \textbf{else} \{c : \mathbb{C} \mid begin \leqslant c\}) \cap$$
$$(\textbf{if } end = \mathsf{null}\ \textbf{then}\ \mathbb{C}\ \textbf{else} \{c : \mathbb{C} \mid c \leqslant end\})$$
[*begin* and *end* define the synchronization span]
$$positions = syncSpan \times (\textbf{if } repeatNumber = 0\ \textbf{then}\ \mathbb{N}$$
$$\textbf{else} \{n : \mathbb{N} \mid n < repeatNumber\})$$
[Positions are always defined by *syncSpan* and *repeatNumber*]
$$\textbf{if } curPosition = \mathsf{null}\ \textbf{then}\ current = \mathsf{null} \wedge repeatCount = 0$$
$$\textbf{else}\ curPosition = current \mapsto repeatCount$$
[Decompose current position to its coordinate and repeat count]
$$direction = \mathsf{forward} \Rightarrow (\_ \prec \_) = (\_\ prec\ \_)$$
$$direction = \mathsf{backward} \Rightarrow (\_ \prec \_) = (\_\ prec^{\sim}\ \_)$$
[Traversal direction defines the order of the positions]

The class aggregates an object of $\downarrow EventDispatcher$. Aggregation allows to separate them in independent processes and to dispatch the events asynchronously. The attributes *repeatNumber* keeps the total number of the traversal loops. The set *syncSpan* defines the span of the synchronization coordinates – The client of a SO may define the boundaries to limit the synchronization to a subset of coordinates. The relation *syncElements* attaches *SyncEvent* objects to certain coordinates. The attribute *positions* defines the positions (pairs of coordinates in *syncSpan* and the repeat counter) that will be passed during traversal. The attributes *curPosition* and *lastPosition* keeps track of the current traversal position, and the position that has just visited last time. The relation $\prec$ defines the order of the traversal according to the specified *direction*. The attributes *positions*, *curPosition* and *lastPosition* are for internal use and hence they are not visible to other objects.

Upon initialization, the repeat number by default is 1 and the repeat count starts from 0. The synchronization state is set to be stopped and there are no *syncEvent* attached to any coordinates. The *syncSpan* covers all the possible coordinates and the traverse direction is forward. The attribute *curPosition* and *lastPosition* are set to null, which means there is no last visited position, and the current position is not given yet.

$$\begin{array}{|l}
\textit{INIT} \\
\hline
\mathit{repeatNumber} = 1 \land \mathit{repeatCount} = 0 \\
\mathit{direction} = \mathsf{forward} \land \mathit{syncSpan} = \mathbb{C} \\
\mathit{syncState} = \mathsf{stopped} \land \mathit{syncElements} = \varnothing \\
\mathit{curPosition} = \mathsf{null} \land \mathit{lastPosition} = \mathsf{null} \\
\hline
\end{array}$$

Synchronization operations. Many operations may cause the synchronization state change and trigger corresponding *StateSyncEvent*. A schema *Transit* is defined to catch this common behavior:

$$\begin{array}{|l}
\mathit{Transit}_0 \\
\hline
\Delta(\mathit{syncState}, \mathit{syncState}') \\
e! : \mathit{StateSyncEvent}_{\circledcirc} \\
\hline
\textbf{if } \mathit{syncState} \neq \mathit{syncState}' \\
\quad \textbf{then } e! \neq \mathsf{null} \land e!.\mathit{source} = \mathit{self} \land \\
\qquad\quad e!.\mathit{oldState} = \mathit{syncState} \land e!.\mathit{newState} = \mathit{syncState}' \\
\quad \textbf{else } e! = \mathsf{null} \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{Transit} \mathrel{\widehat=} \mathit{Transit}_0 \mathbin{;} ([\, e? : \mathit{StateSyncEvent}_{\circledcirc} \,] \bullet \\
\qquad (\textbf{if } e? \neq \mathsf{null} \textbf{ then } \mathit{eventDispatcher}.\mathit{Dispatch}))
\end{array}$$

The operations that manipulate the synchronization state are defined next. The operation *Stop* puts the SO into the stopped state from any other state. Stopping an object also causes its repeat counter, the current position and the last visited position to be reset to their initial state.

$$\begin{array}{|l}
\mathit{Stop}_0 \\
\hline
\Delta(\mathit{syncState}, \mathit{repeatCount}, \mathit{curPosition}, \mathit{lastPosition}) \\
\hline
\mathit{syncState}' = \mathsf{stopped} \land \mathit{repeatCount}' = 0 \\
\mathit{curPosition}' = \mathsf{null} \land \mathit{lastPosition}' = \mathsf{null} \\
\hline
\end{array}$$

$$\mathit{Stop} \mathrel{\widehat=} \mathit{Stop}_0 \land \mathit{Transit}$$

The operation *Ready* gets the stopped SO ready for starting. This operation has no effects if the object is in any state other than stopped:

$$\begin{array}{|l}
\mathit{Ready}_0 \\
\hline
\Delta(\mathit{syncState}) \\
\hline
\mathit{syncState} = \mathsf{stopped} \Rightarrow \mathit{syncState}' = \mathsf{ready} \\
\mathit{syncState} \in \{\mathsf{started}, \mathsf{paused}, \mathsf{ready}\} \Rightarrow \mathit{syncState}' = \mathit{syncState} \\
\hline
\end{array}$$

$$\mathit{Ready} \mathrel{\widehat=} \mathit{Ready}_0 \land \mathit{Transit}$$

A SO can be started when it is ready or paused:

$$\begin{array}{l}
\textit{Start}_0 \\
\hline
\Delta(\textit{syncState}, \textit{curPosition}) \\
\hline
\textit{syncState} = \mathsf{ready} \Rightarrow \\
\quad \textit{curPosition}' = \textit{minimum}(\_ \prec \_, \textit{positions}) \\
\quad \textbf{if } \textit{curPosition}' = \mathsf{null} \textbf{ then } \textit{syncState}' = \mathsf{stopped} \\
\qquad \textbf{else } \textit{syncState}' = \mathsf{started} \\
\textit{syncState} = \mathsf{paused} \Rightarrow \textit{syncState}' = \mathsf{started} \\
\textit{syncState} = \mathsf{started} \Rightarrow \textit{syncState}' = \textit{syncState} \\
\hline
\end{array}$$

$$\textit{Start} \mathrel{\widehat{=}} \textit{Start}_0 \land \textit{Transit}$$

Once the SO has been started, it can be put in to the paused by the operation *Pause* defined below:

$$\begin{array}{l}
\textit{Pause}_0 \\
\hline
\Delta(\textit{syncState}) \\
\hline
\textit{syncState} = \mathsf{started} \Rightarrow \textit{syncState}' = \mathsf{paused} \\
\textit{syncState} \in \{\mathsf{stopped}, \mathsf{ready}, \mathsf{paused}\} \Rightarrow \textit{syncState}' = \textit{syncState} \\
\hline
\end{array}$$

$$\textit{Pause} \mathrel{\widehat{=}} \textit{Pause}_0 \land \textit{Transit}$$

Other operations. "Setter" operations are needed to change the value of the state variables explicitly. These operations are straightforward, nonetheless it should be noticed that because the postcondition of every operation must yield to the predicates in the state schema, changing values of the state variables may implicitly change the value of the others. We omit the specifications (and hereafter) due to the limited space.

A set of event operations need to be defined. For example *SetEventDispatcher* that associates an event dispatcher to the SO, *Attach* and *Detach* operations to add or remove the synchronization events to the coordinate, and *DispatchEvents* takes a sequence of *SyncEvent* sets as input and sends the events simutaneously to the event handler for dispatching.

The progression of a SO is modeled by specifying the coordinates as milestones, unfolding repetition into positions, and defining the visiting order as per direction. The object has now the static basis for dynamic progression. A default "stepping" behavior bring in the dynamic aspects by assuming the next position to visit is the closest position after the current one, without knowing whether it is driven by its own process or by anything else.

A SO has now been defined as such so that no media specific semantics is directly attached to it. For example, there is no notion of time although it is crucial for time-based media objects. Subclasses, realizing specific media control should, through specification, attach concrete semantics to the object through their choice of the type of the internal coordinate system, through a proper specification of what “visiting a position” means, and through a proper specification of how the object should move from the current position to the next.

Other Classes. Other classes depicted in Fig. 1 and Fig. 2, such as *ActiveSynchronizable*, *SyncMediaObject*, *Timer*, *TimedSynchronizable*, *TimedMediaObject*, *EventDispatcher* and *EventHandler*, are omitted here due to the limited space. The complete specifications can be found in [2].

2.4 Consequences

Benefits

Separated timing concerns: Timing constraints are added only when they are necessary and the timer can be shared – which saves concurrent processing resources. Static media objects, for example, static pictures, can be easily implemented as passive SO’s with a simple coordinate system that only has one coordinate and it will stay static until there is an external drive to move it forward – which naturally stops it because there is no next coordinate. If there is a need for the picture to stay presenting itself for a certain amount of time, it can be then implemented as an *TimedSynchronizable* that uses a timer. For media objects that have a progressive presenting behavior and that are not constrained by the time, for example, a TTS object, the *ActiveMediaObject* class does what is needed. However, if the TTS object need to presented at a speed of a video stream, a timer can then be shared to make a *TimedMediaObject* so that the TTS object can speak faster or slower.

Flexible concurrency: The concurrency of the structure can be from none to fully concurrent.

Simple and unified synchronization interface: SO’s at different levels have the same synchronization controlling interface and the same state transition scheme, no matter whether the coordinate system is time based or not. This makes it easy to incorporate new types of SO’s without influencing existing types.

Open for extension: The actual presentation behaviors of media objects are left open, and the event-based synchronization mechanism can be extended for other purposes. For example, user interaction events can be easily added to the structure in addition to the state transition events and the timing events.

Liabilities

Hard to debug: When multiple processes are involved, *Syncronizable* can be hard to debug since the inverted flow of control oscillates between the framing infrastructure and the parallel callback operations on the synchronization controls and event handlers, especially when they are distributed over devices [16].

Event dispatching efficiency depends on the hosting platform and the performance of the network: This is a liability of all event based synchronization mechanisms. In the specification, events are sometimes required to be dispatched simultaneously using a conjunction operator. It is possible to emulate the semantics of the simultaneous operation using multiple threads, however this can only be efficient when the number of the included threads is small.

3 Examples

3.1 DeepSea

In the EU funded project ICE-CREAM we investigated how to make compelling experiences for end-users based on the possibilities of integrating technologies for interactive media, for example, DVB-MHP, MPEG-4, 3D graphics and Internet technologies [17,18]. Technological options that address different levels of interactivity for end-users were investigated and implemented in prototypes. In one of the applications a video is enhanced with fictional content, which uses 3D graphics and animations to enhance the users viewing experience. This application is a prototype of an interactive 3D movie about a deep-sea adventure in a submarine. Aside from the combination of video streams, 3D computer graphics, text, still pictures, soundtracks and multilingual interfaces, it also enhances the user¡¯s experience by distributing the interactive objects to multiple devices (lights, portable displays, robotic toys) to create ambience effects in the performance space - the end user¡¯s environment (fig. 3).

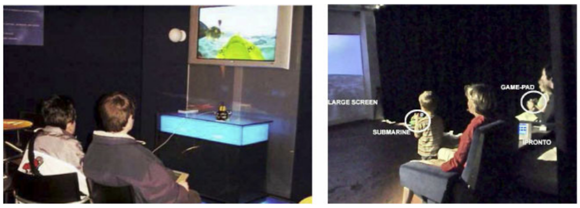

Fig. 3. Distributed media objects in DeapSea

In DeepSea, sound, video objects and 3D animations are *TimedMediaObjects* that have intrinsic timing. Some of the still pictures and text were modeled as static *Synchronizables* that are controlled only by other objects or user interaction, but some of them were *TimedSynchronizabes* because they appear and disappear based on its internal timing mechanism. Lighting effects and robotic behaviors were new forms of *TimedMediaObject* that act and react based on the timing and interaction events. All these objects are synchronized over a network based on an IPML script [13].

3.2 Simulation-Based Delivery Training

The numbers of high-risk pregnancies and premature births are increasing due to the steadily higher age of pregnancy. Medical simulators are used in training the doctors to deal with emergent perinatal situations. To enhance the training effect, sophisticated simulators are integrated into a realistic training environment that takes into account the medical instruments and team aspects. The training environment becomes increasingly complex and requires a clear structure for different training scenarios and flexible hardware configurations. Eindhoven

Fig. 4. Team training with delivery simulators at MMC

University of Technology is cooperating with Máxima Medical Center, aiming at the next generation simulation based training facilities. The result of this effort is the design and implementation of the MedSim system [19]. The system brings software and hardware devices and components into the training room than a single patient simulator, one of which is for example the device designed with audio-visual feedback for medical staff to perform Cardiopulmonary Resuscitation (CPR) more efficiently [20,21]. We aim at an open system architecture that is flexible and extensible enough for the industry to introduce further development and future technologies into simulation based team training. In the design of the delivery simulation system, distributed sensors and actuators are integrated into the mother manikin, the baby manikin, as well as the environment for the purpose of medical team training (fig. 4).

Here the behaviors of manikins are modeled as *TimedSynchronizables* and the projections of 3D graphics are *TimedMediaObjects*. Separated input from the sensors (for example, the positions of the team members detected by the camera) and output from the actuators (skin color and muscle tone of the baby for example) are modeled as *Synchronizables* so that they can be initiated, enabled and disabled at certain moments. The components of the system are coordinated by a scenario based script during a training session [22].

4 Conclusions

The Synchronized Object pattern can be applied for content elements with or without intrinsic timing in distributed multimedia applications. The pattern language is used here to capture the common features of these objects. Although the complete specifications are not included here, the Object-Z examples of a few classes in this pattern are enough to show how a formal method can be applied to treat the architectural construct and compensate the insufficiency of object-oriented language such as UML. The Object-Z specification provides a better amount of generality and abstraction than UML and concrete examples, with the power of both formalization and object orientation for elementary element building blocks – it has the properties that an architectural specification should have [23]. However, the Object-Z specification is not intended to replace natural language specifications. Instead, formal definitions are complementary to existing means (natural language, code samples, etc.). Two examples in gaming and training were briefly introduced without getting into the implementation details, but only showing that the described pattern is applicable for different situations in distributed multimedia applications.

References

1. Alexander, C., Ishikawa, S., Silverstein: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Oxford (1977)
2. Hu, J.: Design of a Distributed Architecture for Enriching Media Experience in Home Theaters. Technische Universiteit Eindhoven (2006)
3. Duke, R., Rose, G.: Formal Object-oriented Specification using Object-Z. Macmillan Press Limited, London (2000)
4. Smith, G.: The Object-Z specification language. In: Advances in formal methods, Kluwer Academic Publishers, Dordrecht (2000)
5. Bartneck, C., Hu, J., Salem, B., Cristescu, R., Rauterberg, M.: Applying virtual and augmented reality in cultural computing. International Joural of Virtual Reality 7(2), 11–18 (2008)
6. Hu, J., Bartneck, C., Salem, B., Rauterberg, M.: Alice's adventures in cultural computing. International Journal of Arts and Technology 1(1), 102–118 (2008)
7. Hu, J.: Move, but right on time. In: Workshop on design and semantics of form and movement (DeSForM 2005), Newcastle upon Tyne, pp. 130–131 (2005)
8. Sullivan, S., Brown, D., Winzeler, L., Sullivan, S.: Programming With the Java Media Framework. John Wiley & Sons, Chichester (1998)
9. Duke, D.J., Herman, I., Marshall, M.S.: PREMO: A Framework for Multimedia Middleware: Specification, Rationale, and Java Binding. LNCS, vol. 1591, p. 1. Springer, Heidelberg (1999)
10. Hu, J.: Distributed Interfaces for a Time-based Media Application. Post-master thesis, Eindhoven University of Technology (2001)
11. Hu, J.: StoryML: Towards distributed interfaces for timed media. In: ten Kate, W. (ed.) Philips Conference InterWebT 2002, NatLab, Eindhoven (2002)
12. Hu, J.: StoryML: Enabling distributed interfaces for interactive media. In: The Twelfth International World Wide Web Conference, Budapest, Hungary (2003)
13. Hu, J., Feijs, L.: IPML: Structuring distributed multimedia presentations in ambient intelligent environments. International Journal of Cognitive Informatics & Natural Intelligence (IJCiNi) 3(2), 37–60 (2009)
14. Duke, D., Duce, D., Herman, I., Faconti, G.: Specifying the PREMO synchronization objects. Technical Report ERCIM-01/97-RD48, ERCIM (1997)
15. Simons, A.J.H.: The theory of classification, part 4: Object types and subtyping. Journal of Object Technology 1(5), 27–35 (2002)
16. Hu, J., Feijs, L.: An adaptive architecture for presenting interactive media onto distributed interfaces. In: Hamza, M. (ed.) AI 2003, pp. 899–904 (2003)
17. Hu, J., Janse, M., Kong, H.J.: User experience evaluation of a distributed interactive movie. In: HCI International 2005, Las Vegas (2005)
18. Feijs, L., Hu, J.: Component-wise mapping of media-needs to a distributed presentation environment. In: The 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), Hong Kong, pp. 250–257 (2004)
19. Hu, J., Peters, P., Delbressine, F., Feijs, L.: Distributed architecture for delivery simulators. In: International Conference on e-Health Networking, Digital Ecosystems and Technologies (EDT 2010), Shenzhen, China, pp. 109–112 (2010)

20. Chen, W., Oetomo, S.B., Feijs, L.M.G., Andriessen, P., Geraets, F.K.M., Thielen, M.: Rhythm of Life Aid (ROLA) – an integrated sensor system for supporting medical staff during cardiopulmonary resuscitation (CPR) of newborn infants. Submitted to IEEE Transactions on Information Technology in Biomedicine (2009)
21. Oetomo, S.B., Feijs, L.M.G., Chen, W., Andriessen, P.: Efficacy of audio-promoted rate guidance for insufflation and chest compressions and feed-back signalling for the pressure of chest compressions during cardio-respiratory resuscitation (CPR) of newborn infants. In: The annual meeting of the Society for Pediatric Research (SPR 2009), Baltimore, US (2009)
22. Hu, J., Feijs, L.: A distributed multi-agent architecture in simulation based medical training. In: Chen, Q. (ed.) Transactions on Edutainment III. LNCS, vol. 5940, pp. 105–115. Springer, Heidelberg (2009)
23. McComb, T., Smith, G.: Architectural design in Object-Z. In: ASWEC 2004: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC 2004), Washington, DC, USA, pp. 77–86. IEEE Computer Society, Los Alamitos (2004)