IEEE International Conference on Multimedia Computing and Systems
May 14.–19., 1994, Boston, USA

Supporting Temporal Multimedia Operations in Object-Oriented Database Systems

Karl Aberer, Wolfgang Klas
GMD-IPSI, Dolivostr. 15, D-64293 Darmstadt, GERMANY
email: {klas,aberer}@darmstadt.gmd.de

Abstract

Advanced applications in fields like electronic publishing and telecooperation face the problem of handling multimedia information. Conventional database systems do not offer adequate support for storage management as they do not provide for the modelling, indexing, and manipulation of multimedia data. Database management systems need to be extended if they should be able to handle multimedia information like audio and video. In this paper we present an approach of extending the data model of an object-oriented database system by means of schedules which allow for the description of the temporal characteristics of multimedia operations as they occur when modelling time-dependent data like audio and video. First, we present a model of schedules and define the basic concepts and semantics to execute time-dependent operations. Second, we introduce the specification language as an extension to the VODAK Model Language and illustrate the concepts by examples. Finally, we briefly discuss the impact of integrating multimedia data modelling support into a database management system on the system architecture.

Keywords. *Object-oriented database systems, temporal modelling, multimedia databases.*

1 Introduction

Advanced applications like electronic journals, news services, and cooperative editing need to deal with multimedia information. The *integrated* handling of video, audio, images, graphics, animations, combined together with textual information in hypermedia documents, become more and more important in such applications. Programming languages, visual interfaces, operating systems, networks, and database management systems need to support the handling of multimedia information. At GMD-IPSI we work on the extension of object-oriented database technology in order to provide integrated database-system services for the management of multimedia data ([9], [12], [13], [15]).

Multimedia information significantly differs from traditional information types like text, numeric data, graphics, and images. In [1] we presented a characterization of the main problems which have to be addressed when trying to provide adequate database management support for multimedia information. There we showed that one of the most important gaps between today's programming and data modelling paradigms employed in the framework of database management systems and the requirements associated with the handling of multimedia data is that appropriate concepts related to time-dependent behavior associated with multimedia data are missing. We derived the necessity of concepts related to time-dependent behavior from the need to provide for scheduling of different tasks on multimedia data, for appropriate simultaneous device interaction, and for efficient handling of user interaction. As a consequence the following support is needed for multimedia data and operations:

- temporal composition of multimedia data,
- synchronization of multimedia operations,
- interruptability of multimedia operations (by users and devices).

The objective of this paper is to define an object-oriented modelling language and the corresponding processing semantics for a database management system that provides such a support. Our goals are

- to provide a solution that is oriented closely to the requirements of modelling time-dependent multimedia data without unnecessary overhead,
- to integrate seamlessly the concepts into the existing data model VML ([8], [10]), with as few extensions as necessary by exploiting existing modelling features of VML,
- to allow for an efficient implementation,
- to achieve a solution oriented towards application development (i.e., macro synchronization), not low level problems (i.e., micro synchronization) with the potential to apply the same concepts to similar problems like workflow management,
- to allow integration with low-level operations provided by database-system services like a continuous object manager at a lower system layer.

In [1] we already discussed that an object-oriented approach is a very promising way to realize a multimedia database system. It allows to capture semantics associated with multimedia data by means of methods defined for classes and their instances. Other possible solutions could be achieved by employing techniques known from the fields of active and real-time database systems. At a first glance, the concept of Event-(Condition)-Action (ECA) rules seems to be adequate for the modelling of time-dependent activities. But the many and quite complex types of events available in the generic ECA approach as well as the lack of concrete timing constraints for rules related to a common time axis does not allow a direct application of it. However, we will be able to provide features similar to those of ECA rules in our approach.

Real-time database systems provide concepts to express timing constraints, but they focus on giving guarantees for real-time behavior of transactions. Instead, we need support for the synchronization of multimedia operations and for expressing temporal relationships of time-dependent data as well as for the specification of quality-of-service parameters (e.g., guaranteed delivery, guaranteed capacity of communication channels).

Timed Petri nets [11] providing the concept of time intervals are an interesting approach and we believe that such concepts are needed for the temporal composition of multimedia data. However, the approach lacks appropriate concepts to deal with the problem of interruptability as well as with conditional activities. Furthermore, the approach provides for the specification of temporal relationships, but no operational model integrated in a data model is available.

In [5] the features of a temporal model for temporal composition of activities are described on top of a conventional object-oriented data model, but these features do not become part of the object-oriented data model and, hence, are not offered by the data model to a system designer. Similar approaches are taken in [16]. However, these models could serve as applications of the concepts proposed in this paper.

[2] discusses interesting solutions at the lower level of communication protocols but is not very suitable for data modelling as required by database systems.

Temporal logic is another very interesting approach which we will use as a general guideline. [3] discusses the integration of the object-oriented paradigm and real-time temporal logic which results in a very powerful specification mechanism. However, a general logical specification language is not oriented towards supporting a feasible operational execution model.

The solution proposed in this paper results in an extension of the open object-oriented VODAK Model Language ([8] and [10]) supporting the modelling of time dependencies. It fits well with the VODAK system-internal multimedia extensions (continuous object manager, multimedia data types, interaction and presentation manager [13][15]). It also allows for an efficient implementation by making use of efficient low level components. Using such low level components is, however, not necessary because of the lack of expressive power of the model, but always represents a compromise between flexible semantic modelling for application design and efficient processing of large amounts of data under critical time constraints.

In summary, the significant features of our approach are as follows:

- Our approach explicitly captures the duration and execution of temporal operations. We distinguish between non-temporal operations, which can be modelled by methods, and temporal operations, which are modelled by the newly introduced concept of schedules. Both are part of the operational interface of objects according to the object-oriented paradigm.
- Events are used to control the execution of temporal operations. Besides a well-defined set of primitive built-in events we introduce two basic events, *Start* and *End* of a temporal operation, and complex events including absolute time modification of the basic events. The primitive built-in events are realized on the basis of signals. Complex events and the *Start/End* events are treated differently and are realized on a logical basis through a scheduler without using signals.
- The whole model is designed in a way that it allows for a concrete execution model and fits well into object-oriented database models and architectures.

The paper is organized as follows. After giving an introductory example in section 2 in order to provide some intuition we present the structural and operational model for the concept of schedules in section 3. In section 4 we propose extensions for the VODAK Model Language VML and give some examples for the usage of the extended language. Finally in section 5 we briefly discuss the consequences on the architecture of the VODAK system.

2 An introductory example

Suppose we want to model a class of presentations in an electronic cinema in an object-oriented database. We assume that in this (simple) presentation first a slide with an advertisement is shown for a fixed period of time, then there is a variable break (for selling articles), where the length of the break is determined by a piece of music played, and then the movie is shown, synchronizing a selected audio and video stream. This scenario will be modelled in the following way (schedules are italicized).

```
CLASS cinema_presentation
PROPERTIES
   advertisement: Slide;
   music: Audio;
   video: Video;
   audio: Audio;
IMPLEMENTATION
   SCHEDULE show(adtime: INT);
     { AT START SELF
         advertisement→display();
       AT START SELF+adtime
         {advertisement→stop_display();
          music→play();}
       AT END  music   video→play();
       AT END  music   audio→play("english");
       AT END video    RETURN; }
END;
```

The schedule *show* consists of a set of schedule statements. Each statement states when (**AT** some event) statements have to be executed. The events in the **AT** clause refer to the start and end of schedules applied to the object referenced in the clause. For example **AT START SELF** refers to the start of the schedule *show* which is executed for the receiver object of this schedule, or **AT END** music refers to the end of the schedule *play* executed for the object music. Additionally, events may be modified by adding a time offset. The actions defined at the different events can either be ordinary method calls (e.g. advertisement→display()) or calls to other schedules (e.g. music→*play*()) or a return statement determining the end of the schedule. Observe also that schedules may be parameterized.

When executing the schedule *show* all actions scheduled for a certain event in the different statements are executed in parallel. Thus when calling subschedules like playing a video and audio, these generates processes running in parallel. The synchronization of the processes is provided by referencing the same events and using a common global clock to determine time offsets.

At a first glance this example seems to be quite trivial, but a closer analysis reveals non-trivial problems, e.g. problems with regard to the event model, synchronization of subschedules, or variable bindings. In the next section we will capture these concepts in a more formal way and develop an execution model that is consistent with the intuition given in this section.

3 Temporal operations in object-oriented databases

3.1 Schedules and events

In this section we discuss the basic concepts of how to introduce the time dimension into an object-oriented data model. The concepts will be applicable to most object-oriented data models, because only a few features of the modelling mechanisms of VML we actually work with are presupposed.

In the VML data model there is a distinction between two types of data, namely objects, which are persistent, and values, which are transient. We assume that multimedia data is always modelled as objects. This is a reasonable assumption as objects are the means to represent persistent data and multimedia data will rarely be available transiently.

VML is a behavioral object-oriented database model. Hence, objects have a well defined interface of operations, which are called *methods*. Objects interact with their environment through messages, which are method calls to objects, while the state of objects is encapsulated. Thus we first discuss the notion of multimedia operations on objects, while the structural aspects related to multimedia data will come in later[1], and aspects related to transient multimedia data and operations on them will not be considered.

The semantics of a method in an object-oriented data model encompasses two characteristics:

- *functional characteristics:* Method executions represent function evaluations that return a value for given parameters. More formally this behavior of a method *m* is described by a function
  $$m: O \times D_1 \times ... \times D_n \rightarrow D,$$
  where O is the set of object identifiers (receiver object) and D_i are the (transient) domains of the parameters, and D is the domain of the result.
- *dynamic*[2] *characteristics:* Method executions represent events that change the state of the database and initiate other events, by issuing messages. Their interdependencies can be described in terms of an event-state diagram. More formally a method *m* changing the state of a database is described by a function
  $$m: O \times D_1 \times ... \times D_n \times DB \rightarrow D \times DB$$
  where DB is the set of possible database states.

For multimedia operations a third characteristics is crucial:

- *temporal characteristics*: Multimedia operations differ from methods as they have typically a duration over time, which is relevant for the specification of the operation as well as for the interdependencies with other operations. An operation with temporal characteristics has a start time, which influences the execution of the

1. Other approaches for multimedia data models ([5],[11]) advocate strongly the temporal composition on multimedia data structures. This can satisfy temporal composition, but not interruptability and synchronization of operations. The opposite can be easily achieved as shown later.
2. The terminology is chosen according to [14]. One must not confuse dynamic characteristics of operations with temporal characteristics.

operation, and an end time, which is determined by the execution of the operation.

Not all operations applicable to objects in a multimedia system need to be considered with their temporal characteristics. Thus we will distinguish between operations with temporal characteristics and those without (despite the fact that also the later needs a concrete interval of time to be executed). This leads to the definition of *schedules*.

Definition: A schedule s in an object-oriented database system is a mapping

$$s: O \times D_1 \times ... \times D_n \times DB \times T \rightarrow D \times DB \times T$$

where O is the set of object identifiers (receiver objects), D_i are the (transient) domains of the parameters, DB is the set of possible database states, and $T=\mathbb{R}$ is the one-dimensional real coordinate axis for time. ■

Schedule signatures are part of the interface of an object (which is typically but not necessarily defined by the object's class). They can have parameters and return values like methods. This comprises the *declarative aspect* of schedule definitions.

The execution of a schedule is initiated like the execution of a method by a message call of the form

receiver_object→schedule_name(parameters).

The execution of a schedule s is related to a *time interval* $[t_{start}, t_{end}] \in T \times T$, $t_{start} \leq t_{end}$ and defines two unique points on the time axis, namely the start of the operation at t_{start} and the end of the operation at t_{end}.

Qualitative temporal relationships between temporal operations can be specified as temporal relationships of the corresponding time intervals $[t_{start}, t_{end}]$ and $[v_{start}, v_{end}]$:

$t_{start} \oplus v_{start}$ and $t_{end} \oplus v_{end}$, where $\oplus \in \{=, \leq, \geq\}$.

These kinds of relationships (of which 13 different exist) are a generalization of the linear ordering relationships for points on the time axis. These relationships are well investigated [7].

In order to refer to the start and end points of temporal operations we introduce the concept of *events*.

Definition: An event E is a countable subset of T, $E \subseteq T$. If $t \in E$ we say that event E occurs at time t. The set of all events is denoted with $\mathbb{E}$. ■

Events are specified by event specifications. In the following we give elementary examples of primitive event specifications that are related to the schedule concept.

- *Start*(o), *End*(o), $o \in O$ are events that correspond to the start or end of an arbitrary schedule executed by an object with object identifier o.
- *Start*(o,s), *End*(o,s) are event expressions that correspond to the start or end of a particular schedule s executed by an object with object identifier o.

Note that for example $Start(o,s) \subseteq Start(o)$.[3] This kind of event is caused only by the execution of schedules. We refer to events of this kind in the following as *internal events*.

Other events that are not related to the execution of schedules are

- *Call*(o), which corresponds to sending a message to an object o,
- *Call*(o,m), which corresponds to sending a particular message m to an object o,
- *Interrupt*(i), $i \in N$, which corresponds to a system interrupt, identified by i.

We will refer to events of this kind as *external events*.

Using *event constructors* new events can be defined from the primitive internal and external events introduced before.

The central event constructor that allows to combine qualitative with *quantitative temporal relationships* is the following:

$$E+t := \{t'+t \mid t' \in E\},\ E \in \mathbb{E},\ t \in T.$$

Other event constructors can also be defined, e.g.

$$E_1 \text{ or } E_2 := E_1 \cup E_2.$$

For further reference on composition of events see e.g. [6].

We assume that a set of *primitive* (*built-in*) *schedules* S_{prim} is given. Besides performing a well-defined task in the system, primitive schedules generate start and end events. Additionally, we introduce a mechanism to define *composite schedules*, S_{comp}.

3.2 Operational specification of schedules

The *operational specification*, or in other words the implementation of composite schedules, is the place where temporal relationships between different operations are specified (similarly as in the implementation of methods the functional and dynamic relationships to other operations are specified).

The specification of a schedule consists of *actions*, that take place during the execution of the schedule. These actions determine the duration of the schedule on the one hand and which operations are executed and when they are executed on the other hand. First, we describe now the possible actions that can take place in the execution of a schedule. The actions are given as sequential programs. The statements that can be used within such programs deter-

3. We will use event specifications to denote the corresponding sets of events without explicitly distinguishing between the specification expression and the event itself.

mine largely the expressive power of schedule concept. The following statement is necessary:

- *Termination statement:* When it is reached the execution of the schedule terminates.

The following statements are considered as a minimal set to obtain reasonable expressive schedules:

- *Schedule calls:* other schedules can be initiated from within a schedule. These are called *subschedules*.
- *Method calls:* these statements allow to integrate any non-temporal operation into the execution of schedules.
- *Conditional statements* (*if–then–else*)*:* these statements give schedules the expressive power of ECA rules.

Additional types of statements, as provided for the implementation of methods, may be added to this list. We denote in the following the set of sequential programs composed of the available statements by $\mathcal{P}$.

In order to obtain a feasible execution model we impose the following restrictions on the elements of $\mathcal{P}$.

- Return values generated by a schedule call must not be used in any other statement of the sequence (R1).
- Recursive schedule calls are not allowed, that is the schedule must not call itself (also not with different parameters) (R2).

As certain statements, like schedule and method calls, return a value, a mechanism for assigning these values is to be provided. Thus we define for each schedule a local scope of variable bindings of type $[v_1: D_1,...,v_n: D_n]$, where v_i are variable names and D_i are domains. We denote with $\mathcal{S}$ the possible scope types.

Now we are ready to define schedule specifications:

Definition: A specification of a *composite schedule* $s \in S_{comp}$ is given by a set of schedule statements $Act_s \subseteq \mathcal{E} \times \mathcal{P}$, and by a scope type $Scope_s \in \mathcal{S}$.

We denote the projection of Act_s on the first component by $Events_s$. This is the set of events on which some action is taken in the schedule execution. ■

Up to now we have not restricted the use of local variables in the specification of composite schedule statements. In particular, local variables could be used for defining events, e.g. by using the + event constructor. However, as the value of the local variables may change throughout the schedule execution, the definition of events in the corresponding schedule statements may also change. We exclude this by the following definition.

Definition: A schedule *s* is *static* when the set $Event_s$ is invariant under execution of schedule statements. ■

In the following we consider only static schedules. We achieve this by excluding the usage of local variables in the event specifications. Still, constants and the values of the parameters may be used in event specifications.[4]

3.3 Execution model for schedules

In this subsection we define the execution model for schedules, which is different from that of methods. The main difference to the execution model of methods comes from the fact that order (and time) of execution of schedule statements can only be determined during runtime. A schedule is initiated at time t_{start} by sending a message

$$v := o \rightarrow s(para)$$

The assignment is only needed when the schedule generates a return value.

There is an important difference between calling a schedule from within a method and from within a schedule. As the interface of a schedule and a method do not differ, a method treats a schedule call like any ordinary method call. I.e. it continues execution not before the return value is received and control is returned by the schedule called (procedural semantics). We call such a schedule call a *root schedule*. A schedule call from within a schedule leads to the parallel execution of the subschedule (process semantics). Restriction R1 on $\mathcal{P}$ is then necessary as the execution of the sequence of statements in a program of $\mathcal{P}$ is continued before the (parallel) subschedule returns a value. Restriction R2 on $\mathcal{P}$ ensures that the calling hierarchy of subschedules form a tree with a root schedule as the root of the hierarchy.

A call of a schedule $o \rightarrow s(para)$ at time t_{start} leads first to processing steps in the database system which we call *registration phase*. In this phase a globally defined registration set $Reg \subseteq \mathcal{E} \times \mathcal{P} \times SID$ is built up. *SID* is a set of schedule identifiers. For a root schedule the set *Reg* is initialized with the empty set. We define as $Sub(s,t)$ the set of all $(e, p) \in Act_s$ for which *e* takes place at time *t* and *p* contains a call to a composite subschedule.

The following recursive procedure performs the registration.

procedure *register*(*o*,*s*)
begin
1. A local scope *scope* of type $Scope_s$ is set up and assigned with default values.
2. All *p*, such that $(e,p) \in Sub(s,t_{start})$ are executed. This leads to a recursive call to *register* for all composite subschedules to be executed at time t_{start} in a subschedule

4. One could also consider actions that e.g. generate new schedule statements, which would lead to a much more complex schedule model.

calling hierarchy before execution of any other operation.

3. $Reg := Reg \cup ((Act_s \setminus Sub(s,t_{start})) \times \{sid\})$, where $sid \in SID$ is a uniquely defined identifier for s. This includes the evaluation of all expressions appearing in the event specifications.
4. The event $Start(o,s)$ is generated
5. Call procedure *execute*()

end

Following the registration phase the system enters the *execution phase* by calling the following procedure.

```
procedure execute()
begin
wait for event such that (event,program,sid) ∈ Reg
    for all (event,program,sid) ∈ Reg do in parallel
        for all statement∈program do in sequential
            if statement=termination_statement
            then remove all elements in Reg with sid;
                 update bindings in local scope of calling schedule with the return value;
                 generate End event of the schedule identified by sid;
            if statement is a method call
            then call message handler;
            if statement∈S_prim then start subprocess;
            if statement∈S_comp then call register;

end
```

Note that we omitted the representation of some of the necessary administration information, e.g. about the structure of the calling hierarchy.

Important remarks:

- The purpose of this execution model is to define the semantics of schedules and not to propose a concrete implementation of processing of schedules. For a concrete implementation still many alternatives are left open. For example, implementing the procedure *execute* directly as described above would require an interpretation of programs $p \in \mathcal{P}$. Obviously in a compiled language like VML a compiler may use functions provided by a central scheduler component for the compilation of p.
- In this model for any $t \in T$ only one registration phase followed by an execution phase takes place. This implies that any execution phase consumes a nonzero amount of time. This also guarantees that all primitive subschedules scheduled for the same event taking place at time t are executed in a single execution phase. This is the central property in order to assure appropriate synchronization of primitive subschedules.
- The execution model shows that the functional and dynamic behavior of schedule execution is inherently non-deterministic, as the order of execution for e.g. for method calls scheduled at the same time is not determined.
- There is an important difference between the treatment of events that are generated by schedules, i.e. start and end of a schedule execution, and thus can be managed internally by the system component for executing schedules, and external events, e.g. system interrupts, that are out of the control of a system component for managing schedules.
- During the execution of a schedule a clock is needed in order to determine composite events that involve time offsets. Conceptually this is the system clock (world time), thus all references to time are synchronized over this clock. In a concrete implementation however the actual clock used may be delayed, or deviate in other ways from the system clock. However, these delays affect the execution of all schedules equally.

Up to now we have only considered the situation where one root schedule is executed at a time. But, in a database system many root schedules will have to be executed simultaneously. In this case there are two alternatives how to treat internal events in the execution phase. Either only the occurrences of those events are detected that are generated in the scope of a particular calling schedule or occurrences are detected globally. In the first case interference between schedules over the database state will still be possible. Thus schedules will not be considered as conventional transactions.

4 The temporal VML data model language

In this section we propose a minimal language extension of VML that allows to define schedules in the form introduced. This is on the one hand to introduce a concrete language for the sake of giving examples and on the other hand to illustrate how the concepts fit into a concrete object-oriented data model language. For simplicity we consider only a reduced version of the VML language.

4.1 Temporal language constructs

In the declarative part of VML we distinguish between methods and schedules. Schedules have signatures in the same way as methods. The syntax for the schedule implementation is given in Figure 1:

```
class_declaration ::=
   CLASS Identifier
   INSTTYPE
      INTERFACE
      [ PROPERTIES   property_definition {property_definition} ]     /* public properties */
      [ METHODS      method_signature {method_signature} ]           /* public methods */
      [ SCHEDULES    schedule_signature {schedule_signature}]        /* public schedules */
      IMPLEMENTATION
      [ METHODS      method_definition {method_definition}]          /* public and private */
      [ SCHEDULES    schedule_definition {schedule_definition}]
   END ";"

schedule_definition      ::= schedule_signature "{" variable_declaration schedule_statement_list"}"
schedule_statement_list  ::= schedule_statement |
                             schedule_statement_list schedule_statement
schedule_statement       ::= AT time_expression statement_list ";"
statement_list           ::= statement | "{" compound_statement "}"
compound_statement       ::= statement | compound_statement ";" statement
statement                ::= RETURN [expression] |
                             if_statement |
                             [variable_identifier":="] receiver_object→schedule_id(parameter_list) |
                             [variable_identifier":="] receiver_object→method_id(parameter_list)
                             variable_identifier":="expression
```

Figure 1: Syntax of schedule implementation

Here we assume that only one schedule call for an object can be executed at a time. Thus we distinguish between an activated state of the object where no other schedule call can be answered, and a silent state. Referencing to the start and end of a schedule executed by an object can then be realized by referring to the object.

```
atomic_time_expression ::=
      START object_identifier |
      END object_identifier
time_expression ::=
      atomic_time_expression ["+" expression]
```

The expression can be any well-formed VML expression returning a real number.

For method implementations a (built-in) boolean method is provided that allows to test whether an object is active or passive:

```
object_identifier→activated();
```

4.2 Illustrating examples

Now we give some examples illustrating how some basic functionalities related to temporal behavior can be expressed using this language.

Periodic behavior. First we show how to model a *periodic behavior*. We do this for an imaginary class animation for animating sequences of pictures. For simplicity we focus on the relevant parts of the class definitions.

```
CLASS Animation
INSTTYPE ...
IMPLEMENTATION
   PROPERTIES
      picture_sequence: Picture_Sequence;
   SCHEDULES
   play_periodic(n: number_of_steps);
      {VAR i: INT;
      AT START SELF
         {i:=1;
          picture_sequence→display_single(i);}
      AT END picture_sequence
         IF i<n
         THEN i:=i+1;
              picture_sequence→display_single(i);
         ELSE RETURN; }
END;
```

```
CLASS Picture_Sequence
INSTTYPE ...
IMPLEMENTATION
  PROPERTIES
    picture: ARRAY[1..n] OF Bitmap;
  SCHEDULES
  display_single(i: INT);
    { AT START SELF picture[i]→display();
      AT START SELF+delta RETURN;
      /* delta is a constant */}
END;
```

Video controllers. We observed that besides specifying temporal relationships between multimedia data and the corresponding operations, another crucial component in multimedia systems is modelling user interaction. We sketch in the following how this can be incorporated by using the language extensions introduced for VML.

```
CLASS Panel
INSTTYPE
INTERFACE
  SCHEDULES
    waitforbutton(): STRING;  /* built-in
    schedule, terminates when button pushed */
END;
```

```
CLASS Controller
INSTTYPE ...
IMPLEMENTATION
  SCHEDULES
  control(v:Video, p:Panel);
    { VAR  b: STRING;
    AT START SELF b:=p→waitforbutton();
    AT END p
        { IF b=="end" RETURN;
          IF b=="start" THEN  v→play();
          IF b=="stop" && v→activated()
            THEN v→stop();
          IF b<>"end"
            THEN b:=p→waitforbutton(); } }
END
```

Temporal relationships. Up to now we have introduced ways to specify temporal relationships between multimedia operations. We will show now how to exploit this specification mechanism in order to perform temporal composition of multimedia data. We give here examples of how sequential and parallel composition of multimedia data can be modelled.

```
CLASS Time−dependent−data
INSTTYPE
INTERFACE
  PROPERTIES duration: INT:
  METHODS stop(); pause(); goto(t: INT);
  SCHEDULES play();
END;

CLASS Sequential−composition
OWNTYPE  /* class methods */
  METHODS
    create(d1, d2: Time−dependent−data):
        Sequential−composition;
INSTTYPE
INTERFACE
  PROPERTIES d1, d2: Time−dependent−
data;
  METHODS
    duration(); stop(); pause(); goto(t:INT);
  SCHEDULES play();
IMPLEMENTATION
  METHODS
    duration(): INT;
      {RETURN d1.duration+d2.duration)};
    stop();
      {IF d1→activated()  THEN d1→stop()
                          ELSE d2→stop();}
    goto(t: INT);
      {IF t<d1→duration() THEN d1→goto(t)
                         ELSE d2→goto(t)};
  SCHEDULES
    play() {AT START SELF d1→play() ;
            AT END d1 d2→play();
            AT END d2 RETURN ;};
END;

CLASS Parallel−composition
OWNTYPE /* class methods */
  METHODS
    create(d1, d2: Time−dependent−data):
        Parallel−composition;
INSTTYPE
INTERFACE
  PROPERTIES d1, d2: Time−dependent−data
  METHODS
    duration(); stop(); pause(); goto(t:INT);
    maxtime(): Time−dependent−data;
  SCHEDULES play();
IMPLEMENTATION
  METHODS
    duration(): INT;
      {RETURN max(d1.duration,d2.duration)};
    stop();
      {d1→stop(); d2→stop();}
```

```
    goto(t: INT);
      {d1→goto(t); d2→goto(t);};
    maxtime(): Time−dependent−data;
      {IF d1→duration()>d2→duration()
      THEN RETURN d1
      ELSE RETURN d2;}
  SCHEDULES
    play()
      { AT START SELF d1→play();
        AT START SELF d2→play() ;
        AT END SELF→maxtime() RETURN ;};
END;
```

A parallel composition of an audio with a video is then specified as follows:

```
VAR presentation, audio,
    video: Time−dependent−data;
presentation:=
      Parallel−composition→create(audio,vid-
eo);
```

Using the metaclass mechanism of VML [10] one can use the above technique to introduce this composition mechanism in the form of a data model extension.

5 Implementation and architecture:

For the implementation and integration of the extensions proposed for VML in VODAK the following requirements have priority for the development of a database component for managing schedule execution, which will be called *schedule manager* in the following.

Use on high granularity: The modelling mechanism itself is powerful enough to specify temporal relationships on a low granularity, e.g. single video frames or audio samples, as well as high level composition of multimedia documents and user interaction. The latter is what we consider the relevant application for the proposed extensions of a database programming language, while the processing at low granularity needs high performance processing of data streams which are features we assume to be available as primitive schedules.

In VODAK currently different built-in data types for efficient processing of audio and video data together with highly parametrized operations (Quality of Service parameters) and components for user interaction [15] are provided. In order to capture the temporal characteristics of operations defined for these data types, they will be made visible in the data model in form of built-in classes together with appropriate primitive schedules. The built-in operations on multimedia data types synchronize themselves with the system clock in the same way as all composite schedules do. Therefore synchronization of application-defined composite schedules with built–in primitive schedules is guaranteed.

Compatibility with current architecture: VML is the data model for an existing DBMS, namely VODAK. Thus, we have to be compatible with the existing components, in particular with the message handler and transaction manager. As it can be seen from the operational model for schedule processing the only interface needed to the DBMS is through method calls. The message handler has to be adapted in a way, that it recognizes schedule calls which have to be delegated to the schedule manager.

Rapid prototyping: In order to prove the usability of the concepts introduced a major concern is to provide prototypes as rapidly as possible. Thus, a first prototype will be based on a single process schedule manager following closely the conceptual execution model explained above.

6 Conclusions

We have introduced an object-oriented data model which supports the description of the temporal behavior of multimedia operations and data. We have introduced the concept of schedules which are either built-in temporal operations, like playback of audio and video streams, or are user-defined composite operations, that can specify temporal behavior using an event concept and references to a system clock. We have given an operational semantics for the execution of schedules and showed how to extend a concrete language with a schedule construct. Finally, we have sketched the integration of this concept into the current architecture of VODAK which is currently under way in the AMOS project (e.g., [12],[13],[15]).

The model introduced offers different alternatives at the design level, e.g. the event model or scope definitions, as well as at the implementation level. For further work we will foremost explore these alternatives, like more complex event models, including support for qualitative relationships (e.g. before and after) and composition operators for events, or different implementation strategies, in particular with regard to the usage of parallel execution of processes.

7 Literature

[1] Aberer K., W. Klas: The Impact of Multimedia Data on Database Mangement Systems. *ICSI, TR-92-065*, Berkeley, Ca., September 1992.

[2] Anderson D.P., L.Delgrosse, R.G.Herrtwich: Structure and Scheduling in Real-Time Protocol Implementations *ICSI, TR-90–021*, Berkeley, Ca., June 1990.

[3] Böhm K., A.Sernadas: An Institution of Real-Time Object Behavior. Technical Report No.4/93, Department of Mathematics –IST (Lisbon Institute of Technology), March 1993.

[4] Dayal U., A. Buchmann, U.Chakravarthy, M.Hsu: The HiPAC Project: Combining Databases and Timing Constraints. In SIGMOD RECORD 17/1, March 1988.

[5] Gibbs S., C. Breiteneder, D. Tsichritzis: Audio/Video Databases: An Object–Oriented Approach. Object Frameworks, Université de Geneve, 1992, pp. 275–291. *Proc. of IEEE Ninth International Conference on Data Engineering* (Vienna, April 1993), IEEE, Los Alamitos, 1993.

[6] Gehani, N.H., Jagadish, Shmueli: Composite event specification in active databases: Model & Implementation. *Proc. of the 18th Int. Conference on Very Large Databases,* August 1992.

[7] Hamblin C.L.: Instants and Intervals. In J.T.Frase et al.(Eds.): Proc. 1st Conf. Int. Soc. for the Study of Time, Springer 1992.

[8] Klas W. et al.: VML – The VODAK Model Language Version 3.1, *Technical Report, GMD-IPSI,* 1993.

[9] Klas W.: Tailoring an Object-Oriented Database System to Integrate External Multimedia Devices. *International Workshop on Heterogeneous Databases and Semantic Interoperability,* Boulder, February 1992.

[10] Klas W., K.Aberer, E.J.Neuhold: Object–Oriented Modelling for Hypermedia Systems using the VODAK Modelling Language (VML). To appear in: A.Biliris, T.Ozsu (Eds.): Object-Oriented Database Management Systems. NATO ASI Series, Springer Verlag Berlin Heidelberg, Dezember 1993.

[11] Little T. D. C., A. Ghafoor: Synchronization and Storage Models for Multimedia Objects. IEEE J. Select. Areas Commun., Vol. 8, No.3, 1990.

[12] Rakow T. C., P. Muth: The V^3 Video Server – Managing Analog and Digital Video Clips. *SIGMOD '93, Washington DC,* May 1993

[13] Rakow T.C., M. Löhr, F. Moser, E. J. Neuhold, K. Süllow: Einsatz von objektorientierten Datenbanksystemen für Multimedia-Anwendungen (Using Object-Oriented Database Systems for Multimedia Applications). it + ti – Informationstechnik und Technische Informatik, Themenheft *Multimedia/Hypermedia*, Teil 2, Oldenbourg Verlag, München, Juni 1993.

[14] Rumbaugh J.E., M.Blaha, W.Premerlani, F.Eddy, and W.Lorensen: Object-oriented Modelling and Design. Prentice-Hall, 1991.

[15] Thimm H., T. C. Rakow: Upgrading Multimedia Data Handling Services of a Database Mangement System by an Interaction Manager. *Technical report GMD* No. 762, Sankt Augustin, July 1993.

[16] Woelk D., W. Kim, and W.Luther: An Object-Oriented Approach to Multimedia Databases; *ACM SIGMOD Record 1986*, pp. 311 – 325, ACM, 1986.