MML: A Language for Modeling Interactive Multimedia Applications

Andreas Pleuß
Ludwig-Maximilians-Universität München
Department for Computer Science, Media Informatics Group
andreas.pleuss@ifi.lmu.de

Abstract

The development of highly interactive multimedia applications is still a challenging and complex task. In addition to the application logic multimedia applications typically provide a sophisticated user interface with integrated media objects. As a consequence, the development process involves different experts for software design, user interface design, and media design. There is still a lack of concepts for a structured development process to integrate these requirements.

In this paper we introduce the Multimedia Modeling Language *(*MML*), a visual modeling language supporting the design process in multimedia application development. It is part of a model-driven development approach for multimedia applications. The language is oriented on well-established software engineering concepts, in particular UML 2.0. It integrates the results of two different research lines: application-oriented multimedia modeling and model-based user interface development. In this paper we describe the concepts of the language and present the modeling process with MML. In particular we show how MML aims to integrate the different developer roles in multimedia application design.*

1 Introduction

The development of multimedia applications is still a challenging task requiring much time and effort. This holds especially for highly interactive multimedia applications which integrate media objects and complex user interfaces with an amount of application logic comparable to conventional (e.g. business) applications. Examples for this type of application are traditional multimedia applications – like games, training, or simulation software – but more and more also information systems with intensive usage of media. An example are so-called "infotainment systems" in cars, which provide the user the control over the car's complete edutainment and comfort functionality through a multimodal multimedia user interface.

Development support typically used for such applications are multimedia authoring tools like *Flash* or *Director*. However, there is still a lack of the integration of principles from software engineering to achieve a more structured and efficient development, as observed by various research work like [1], [13], [2]. The main problems lie in the low support of the authoring tools to structure the applications and in the lack of an adequate software design phase which integrates all aspects of a multimedia application.

On of the most successful state-of-the-art concepts in conventional software development is the usage of visual modeling languages. They are an excellent tool to support the application design phase. In addition, modeling languages can be used for the automatic generation of code skeletons from the models. This can help to overcome the problem of the code structuring in authoring tools as expert knowledge, how to achieve a better overall structure of the code, can be put into the code generator. However, existing modeling languages like the de-facto standard UML [10] are not sufficient for multimedia application development as they do not cover neither media integration nor general user interface aspects [6], [4], [16].

In this paper we propose the *Multimedia Modeling Language* (MML), an abstract and platform-independent visual modeling language for interactive multimedia applications. It bases on the concepts of UML 2.0. MML is intended to support a design phase for the overall application. In particular, it aims to integrate the involved different developer teams for software design, media design and user interface design.

MML integrates the results of two different research lines. The first one are the application-oriented modeling approaches for multimedia and hypermedia applications. For hypermedia applications, a large spectrum of modeling approaches already exist, e.g. *UWE* [8], *OO-H* [5] or [9]. However, they are optimized for HTML-based applications. Even if they support the integration of media objects into websites, they do not cover the highly interactive and dynamic character of a multimedia application user interface. A multimedia-specific approach is *OMMMA* (Object Oriented Modeling of Multimedia Applications, [14]). On that base we propose in [7] the overall framework for a model-driven development of multimedia applications. The approach uses models for the application *structure* and the coarse-grained overall behavior (in terms of *scenes*) as well as simple models for the *user interface presentation* and the *interaction*.

The second important research line is model-based user interface development. An overview over the various work developed in this field is provided e.g. in [15] and [16]. Common concept in this field is the definition of an *abstract user interface model* which contains *abstract user interface elements*. The abstract user interface elements are assigned to *presentation units*, which are abstractions of a window on a graphical user interface. On that base, an *interaction model* is provided. Often the abstract user interface and the interaction are derived from a *task model* [11] which is specified during requirement analysis.

As the user interface plays a key role in multimedia applications, it should be obvious to consider the knowledge from user interface modeling. In [12] we discuss the general concepts and requirements for the integration of user interface modeling and model-driven multimedia application development. In the current paper we propose with MML a concrete solution for those concepts by integrating them into the overall structure derived from [7].

Throughout the paper we use a break-out game as running example. We believe that games are very well-suited as examples for multimedia applications. On the one hand they emphasize on the typical characteristics of multimedia applications like a sophisticated and highly interactive user interface and the heavy usage of multimedia. On the other hand, their expected functionality is relatively easy to understand and does not require specific domain knowledge. A screenshot of the example break-out game is given in figure 1: The player can steer the paddle horizontally to keep the ball within the playing field. When the ball moves off the field the player looses one of his lives. A brick is removed from the playing field if it is hit by the ball. The player reaches the next level of the game, when all bricks are removed from the field. The game is over when the player has lost all his lives.

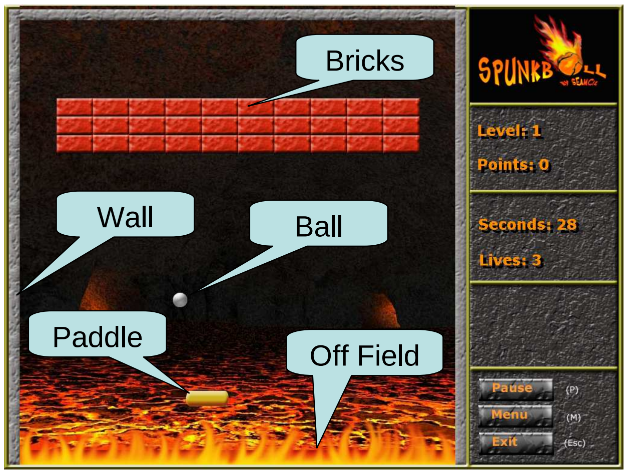

Figure 1. An example break-out game implementation

The rest of the paper is structured as follows: MML is introduced by presenting the structural model (section 2), the scene model (section 3), the abstract user interface model (section 4), and interaction model (section 5). On that base we propose a design process for multimedia applications considering the integration of software designer, media designer and user interface designer (section 6). Finally, we provide the conclusions and an outlook (section 7).

2 Application Structure

In this section we describe the model of the static application structure. As explained in [7] we use the well-established concepts of UML 2.0 class diagrams to model the structure of the application logic in an object-oriented and platform-independent way. Therefore we provide all constructs from UML 2.0 class diagrams, e.g. class properties like attributes, operations, and associations, as well as class relationships like generalizations. Usually, a class represents an entity of the multimedia applications program logic (referred to as *application entity*), like `Player`, `Ball`, or `Paddle` in the break-out game example.

In addition, fundamental parts of a multimedia application are the media objects. In [7] we propose to model them as specific kind of a class and include them in the class diagram. In particular, they are related to the application entities: e.g. in our example the animation which represents the ball is related to the program logic of the ball (i.e. the application entity `Ball`).

Furthermore, [7] defines the media elements as *media components* which encapsulate the media document itself as well as the required functionality to present it to the user. For example the specification of a video in the model implies the ability to encode and play the video within the application.

In [12] we observe the need to specify the different media types within the modeling language itself instead of specifying them within the models. This is necessary as they differ regarding their inner structure (see below) and their possible behavior on the user interface (see section 4). In addition, [12] explains the requirement to include the inner structure of media objects in the model if it is related with application code: For example in a racing game there is an animation representing the racing car. It includes inner (sub-) animations for the wheels and the door, which can be moved independently. As wheels and doors have then to be accessed by the application code, they must be explicitly specified in the model. The possible inner objects of a media object depend on the media type (see [12]).

Figure 2 shows a MML class diagram for the break-out example. Beside the standard UML elements for the application entities (e.g. `Ball`) it contains media components (here denoted in grey color, e.g. `BallAnimation`) which are branded with a keyword for the respective media type (e.g. `Animation`). A media component can provide attributes and methods like an ordinary class.

An application entity can be represented by one or more media components. This is specified by the relationship *MediaRepresentation*, denoted as a dashed arrow, like between `Ball` and `BallAnimation`. To model that a media component represents only a specific attribute of the application entity, the MediaRepresentation can be annotated with the respective attribute name. A media component can be instantiated multiple times within the application, like e.g. the `BrickAnimation` in the break-out game. This is modeled by attaching a multiplicity to the MediaRepresentation. An associated optional keyword specifies whether all instances behave exactly identical (keyword `ref` for *reference*) – i.e. they all reference the same single internal instance – or whether they may behave different (keyword `copy`).

A media component can contain inner objects (e.g. for `LevelGraphic`). Inner objects are denoted within the rectangle which represents the media component. It is possible to decompose them hierarchically into a tree structure with the media component itself as root node. The edges leading to an inner object can be annotated with multiplicities analogous to the MediaRepresentation relationship.

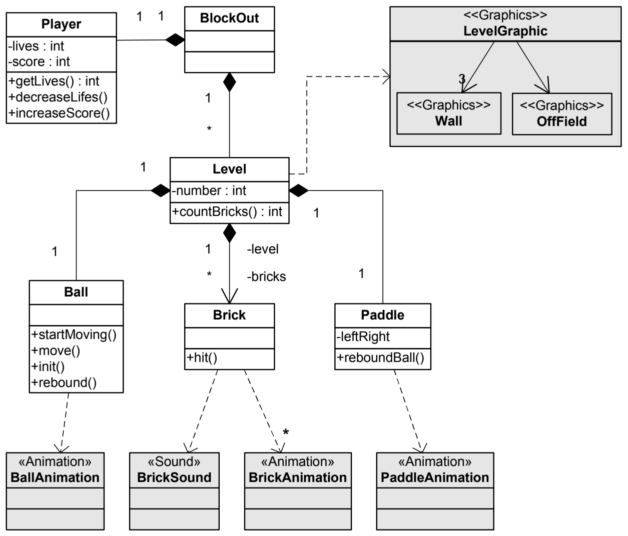

Figure 2. The structure model for the break-out game implementation

3 Scene Model

A common concept in user interface modeling is the *presentation units*. Presentation units are containers for the user interface elements. They can be seen as an abstraction of a window of a graphical user interface. Like conventional applications, multimedia applications usually also show different presentation units to the user. A difference to conventional applications is caused by the time-dependent behavior of media elements: they add an internal state to the presentation unit. For example, when a presentation unit presenting a video is interrupted while the video is playing (e.g. to show a help window) the video should possibly be continued after the interruption.

In [7] we address this requirement by *scenes*. A scene represents a state of the application which corresponds to the presentation of a specific presentation unit. In addition, a scene may have attributes and operations to realize its internal state. Thus, a scene is also a specific kind of a class and can be seen as a controller of the application (in terms of the Model-View-Controller pattern). In particular, scenes own specific operations called *entry-operations* and *exit-operations*. An entry-operation is used to initialize the scene. A scene is always invoked by the call of an entry-operation. An entry-operation can be branded with the keyword `history` which denotes that the operation resumes the last state of the scene which it had when it was active before. Exit-operations are used to leave the scene and to proceed to another scene.

We use the scenes to specify the overall behavior of the application. Therefore we use an adapted kind of state chart. The states correspond to the scenes of the application. A transition between two scenes implies the call of an exit-operation in the outgoing scene and the call of an entry-operation in the target scene. As scenes may provide several entry-operations, each transition is annotated with the name of its target entry-operation.

Figure 3 shows the scene model for our break-out example. The application starts by invoking the `Menu` scene through the entry-operation `initialMenu()`. From the menu it is possible to reach the `Help` scene or the `Game` scene. The `Game` scene is invoked through the entry-operation `startGame`, which holds parameters containing the setting from the menu, i.e. a `Player` object and a boolean value indicating whether sound should be on or off during the game. After a level of the game is finished the application shows the current *Score*. If the player has some lives left it is possible to play the next level. Otherwise the application presents the current `Highscore` list and then resumes to the menu. When the application returns to the `Menu` scene it calls the entry-operation `resumeMenu()`, which is branded with the keyword `history` to indicate that its internal state is resumed.

From the scene diagram, the class properties of the scene can be derived. The entry-operations are visible in the diagram. The parameters of the entry-operations indicate the attributes of the scene. The exit-methods are usually not shown in the diagram, as they are derived from the transitions between the scenes. The implicit entry-methods then get a default name with the prefix `exitTo` followed by the name of the target scene (e.g. `exitToMenu`). To provide an explicit overview over the scene's properties it is optionally possible to additionally show them in the class diagram as specific classes labeled with the keyword `Scene`.

To increase clarity, it is often useful to explicitly model the scenes in terms of classes in the class diagram.

4 Abstract User Interface

In this section we describe the abstract user interface model derived from existing user interface modeling approaches and its integration into MML. Thereby we consider the results of [12] to integrate the media objects and the abstract user interface elements. As in MML scenes act as presentation units (see section 3) all user interface elements are assigned to one of the scenes from the scene model.

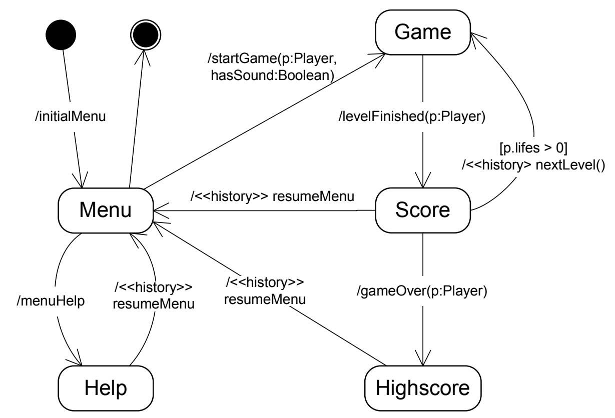

Figure 3. The scene model for the example application.

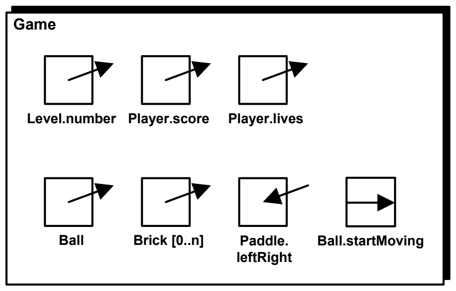

Figure 4. The abstract user interface for the scene `Game`

We use in MML the abstract user interface elements proposed in [17]. Three kinds of elements from [17] are relevant for our purpose: *outputComponents*, which provide some information to the user, *inputComponents*, which allow the user to provide data input, and *actionComponents*, which allow the user to invoke an action of the application (without additional data input). Usually the inputComponents act simultaneously as outputComponents, as they also show the provided input to the user. All kinds of abstract user interface components can be associated with information from the structural model, e.g. an outputComponent mostly represents an attribute of an application entity.

Figure 4 shows the abstract user interface model for the `Game` scene. It contains three outputComponents (denoted by a rectangle with an outgoing arrow) for the player's status information: `Level.number`, `Player.score`, and `Player.lives`. The name of the component indicates the associated information of the structural model, i.e. the attribute `number` of the class `Level`. Formally, this means that the user interface component has a relationship (called *UIRepresentation*) to the respective element from the structural model. In addition it is optionally possible to model the UIRepresentations explicitly in the diagram by adding the respective application entities to the diagram.

A user interface component may also represent information from the structural model which consists of multiple objects or values. This is specified by the assignment of a multiplicity to the respective UIRepresentation. If the UIRepresentations are omitted in the diagram, the multiplicity can be denoted after the name of the user interface component, like for the outputComponent `Brick` in figure 4.

The ball is represented by an outputComponent `Ball`. As the user controls the paddle, the scene provides an inputComponent (denoted as a rectangle with an ingoing arrow) `Paddle.leftRight`. Furthermore, it contains an actionComponent (denoted as a rectangle containing a right-pointing arrow) `Ball.startMoving` to allow the user to start the game.

In [12] we discuss in detail that the media components may partially realize the abstract user interface elements. The realization of an abstract user interface element through a media component is specified in MML by the *AUIRealization* relationship denoted by a dashed arrow (see figure 5). In addition it is also possible that a media component realizes the presentation of the scene itself, like in our example the `LevelGraphic`.

Furthermore [12] shows that media types can invoke actions themselves. An example is a time-related event which is triggered by a temporal media object, e.g. by a video after it has finished playing. Another example is an animation which triggers an event when it collides with another object on the screen or when it reaches a specific screen region. To specify the observation of such events we introduce so-called *sensors*. We model them in MML as a specific kind of UML *AcceptEventActions*. Figure 5 shows four collision sensors (branded with the keyword `collision`) provided by the `BallAnimation`. They observe whether the ball animation collides with a target object. The respective target object is specified by a relationship denoted with the keyword `test`.

In summary, the abstract user interface model has to be enhanced with the information which media components realize user interface components as well as with sensors. If the enhanced model should be differed from the purely abstract user interface model it can be referred to as *media user interface model*.

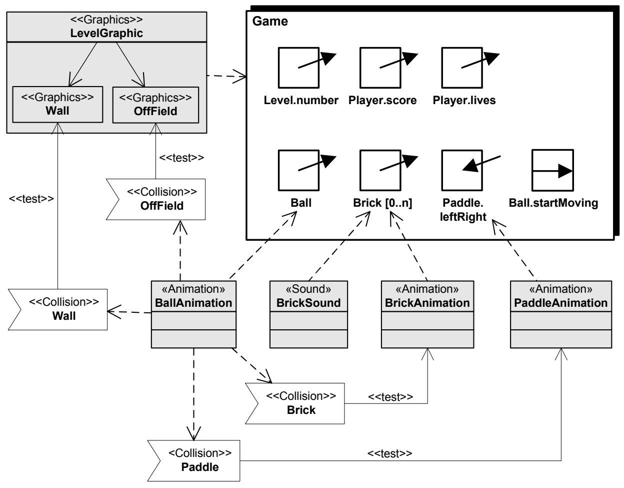

Figure 5. The integrated media user interface model.

5 Interaction

The interaction model describes the behavior of the application under consideration of the user input. MML uses therefore an extended kind of UML activity diagrams, as proposed in several user interface modeling approaches like [17], [3]. They differ from plain UML in the integration of the abstract user interface objects, which may be associated with one or more actions. For our purposes, it is additionally necessary to include the sensors provided by media objects in the diagram, as they also directly influence the application behavior.

Furthermore, we specify more precise rules for the usage of activities and their contained actions to support clearer models and to enable code generation. In MML, the interaction of each scene is modeled by exactly one activity (which may contain sub-activities). All actions in MML are UML *CallOperationActions*, i.e. they correspond to the call of an operation. The operations belong either to the scene itself or to a class from the structural model, which can be an application entity or a media component. According to UML 2.0 the class name can be specified in brackets below the operation name. For example in 6 the first action after the start of the activity calls the operation `init` from the class `Ball`, e.g. placing the ball on its start position.

The inputComponents and the actionComponents influence the application's behavior when the user provides input or triggers an action. Thus, we use them like UML *AcceptEventActions* which wait for an event

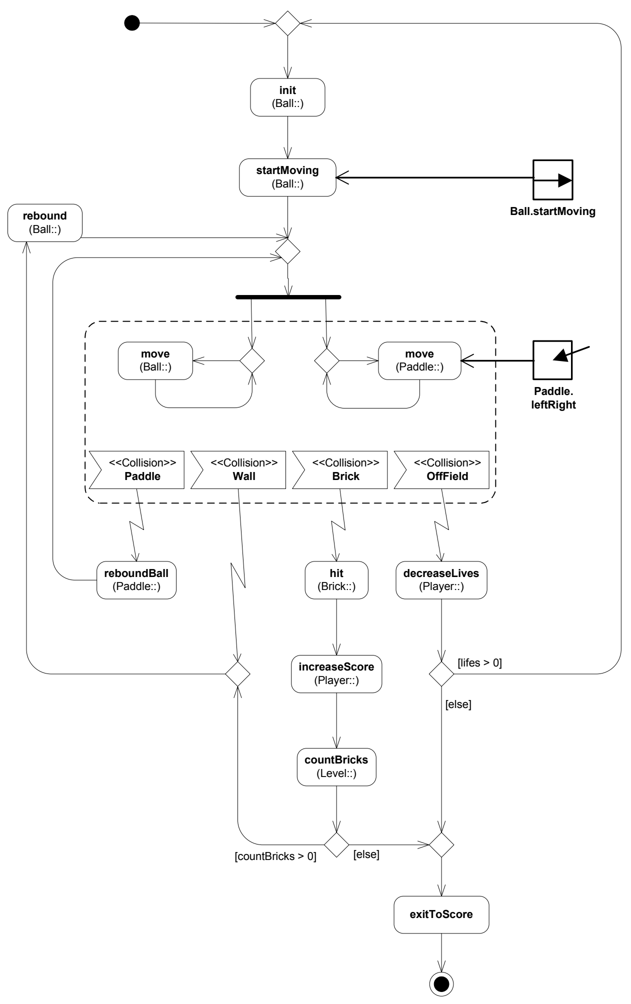

Figure 6. The interaction model for the scene `Game`

– here the associated user action – and offer tokens to their outgoing edges when the event occurs. In our example the actionComponent `Ball.startMoving` has an outgoing edge to the `startMoving` action. Like in UML, the actions are performed when they receive tokens on all their ingoing edges (i.e. when all their foregoing actions are completed). As `startMoving` has two ingoing edges, the action is performed when `initBall` has been completed *and* the user has invoked the actionComponent `Ball.startMoving`.

The outputComponents are used like UML *SendEventActions*. They are implicitly updated after the information they represent has changed. Often it is not necessary to specify this explicitly. Thus, the integration of the outputComponents is optional in the MML interaction model.

The sensors from media components can be used like conventional UML *AcceptEventActions*. In the example in figure 6 we use them to leave an UML *InterruptibleActivityRegion* (denoted by the dashed rectangle) via an UML *interruptible activity edge* (denoted by a lightning-bolt edge).

We show the MML interaction model by a comprehensible example to demonstrate the suitability also for non-trivial behavior as it typically may emerge for an interactive multimedia user interface. Figure 6 shows the interaction model for the `Game` scene from the break-out example. The activity starts with the initialization of the ball (`init`). When the user invokes the `Ball.startMoving` actionComponent, the ball starts to move (`startMoving` action). In the following, the ball moves continuously over the playing field (`move` operation from class `Ball`). In parallel, the paddle moves (`move` operation from class `Paddle`) every time when the user performs an input on the `Paddle.leftRight` inputComponent. Both actions are interrupted, when one of the collision sensors detects a collision of the ball. In case of a collision with the wall the ball is simply rebounced (`rebound` operation of the class `Ball`) and continues to move. When the ball collides with the paddle it is rebounded according to the `reboundBall` operation of the class `Paddle` as the paddle usually is used to control the ball direction (e.g. by giving it spin). When the ball hits a brick, the brick disappears (`hit`) and the player's score is increased (`increaseScore`). Afterwards the number of the remaining bricks is tested. If there are no bricks left, the scene's exit-operation `exitToScore` is executed and the activity terminates. Similar, the number of the player's lives is decreased when the ball leaves the playing field (`decreaseLives`) and the activity terminates when no more lives are left.

The activity diagram describes the complete interaction of the `Game` scene. If required, the UML structuring mechanisms can be used to decompose the diagram in several sub-units (i.e. sub-activities). As the abstract user interface elements and the sensors from media elements are mapped to UML constructs, the whole semantic of the model is compliant to the UML specification. We achieve a high degree of formalization by restricting the actions to operation calls. Also the constraints in guards (e.g. `lives > 0`) may only refer to attributes or operations from the structural model. In addition, the actions leading to a final node must always refer to exit-operations of the scene (`exitToScore` in the example).

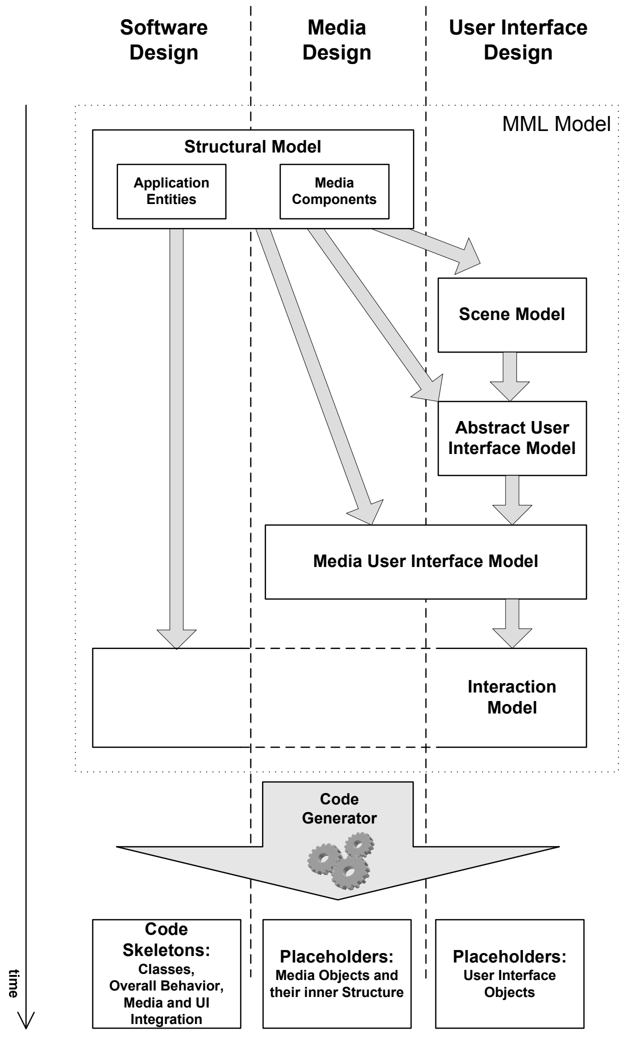
Software Design
Media Design
User Interface Design
MML Model
Structural Model
Application Entities
Media Components
Scene Model
Abstract User Interface Model
Media User Interface Model
Interaction Model
Code Generator
Code Skeletons: Classes, Overall Behavior, Media and UI Integration
Placeholders: Media Objects and their inner Structure
Placeholders: User Interface Objects
time

Figure 7. The application design process with MML

6 Design Process

In the foregoing section we present how to model the different aspects of a multimedia application with MML. MML aims to support a structured application design, building the bridge between the requirement specification and the implementation. The diagram in figure 7 shows the design process. In the following we first explain the aspects shown in the diagram: the assignment of models to one of the developer groups which take part in the development process and the refinement of models during the process. Later in this section we go stepwise through the diagram.

A core characteristic of multimedia applications is the involvement of experts for application design, user interface design and media design. The abstract application model helps to integrate the results from the different developer groups in a consistent way. It can act as a common communication base and also as a kind of contract. Therefore it is mandatory to assign (as far as possible) each part of the model to a responsible developer group. This clarifies the responsibility for the respective part of the model but also ensures that the available knowledge is put into the right places within the model. Thus, the diagram in figure 7 is vertically subdivided in three sections which assign the assets of the design process to the responsibility of software design, media design or user interface design.

The main parts of the design process in figure 7 are the models we explained in the sections 2 to 5. The arrows in figure 7 show how the models build up on each other. If a model builds up on a foregoing model, the development of the subsequent model usually leads to refinements on the foregoing model, e.g. the supplementation with some elements. Thus, the arrows between the models in figure 7 denote on the one hand the derivation of the models but on the other hand they also imply the refinement of foregoing models. Thus, in the following description of the design process we also provide rules for the refinements in each development step.

The base of the design process is the requirement specification. The requirement specification may include use case diagrams for the software related requirements and a task model for the user interface related requirements of the application. Several user interface modeling approaches propose how to derive from a task model the presentation units (i.e. here the scenes), the abstract user interface components, and the interaction model (see e.g. [3]). The derivation of the application entities can be performed like in conventional software development.

The core of MML models is the structural model (see section 2) which can be decomposed into application entities as part of the software design and media components as part of the media design. On that base the scene model (section 3) is specified. It is part of the user interface design. From the structural model, mainly the identified media objects may influence the choice of the different scenes. The scene model may lead to refinements in the structural model: on the one hand the scenes may be added to the class diagram; on the other hand they may require additional media components for the representation of themselves.

The abstract user interface model (section 4) defines for each scene the associated abstract user interface elements. It is part of the user interface design. The abstract user interface elements are related to application entities in the structural model. Thus, their specification helps to detect missing information in the class diagram, i.e. the need of an additional property of an application entity. Besides, actionComponents (e.g. a cancel button) may imply additional transitions in the scene model.

The integration of the media components into the abstract user interface (also described in section 4, here referred to as *Media User Interface Model*), is derived from the abstract user interface and the media components in the structural model. Here, the user interface designer and the media designer should work together.

Finally the interaction model (section 5) is specified using the elements from the media user interface model and the structural model. During the specification of the interaction model missing operations in the structural model as well as missing abstract user interface elements may be detected. The basic steps of the interaction are modeled by the user interface designer. However, the final formal definition in terms of operations from classes of the structural model is also a task of software design. The media designer is involved in this task, if the scene's behavior is influenced by media objects, like in the example. But this does not apply for all scenes in multimedia applications: e.g. the behavior of the `Menu` scene may be specified independently from media objects.

The overall MML model allows the automatic generation of code skeletons as described in [12]. In summary, we aim to generate code for those parts of the application which are difficult to implement manually. Those are mainly the overall structure of the application and the integration of the various media and user interface objects into the application logic. However, we omit code generation for those parts of the application which are cumbersome to define completely in the model – like the implementation of the class operations – or which are traditionally better performed manually, like the concrete user interface realization and the media production. However, for those parts we generate placeholders to be just filled out in the authoring tool or development tool of the respective target platform (e.g. Flash).

A very abstract view on the integration of application logic, media elements, and user interface is provided in figure 8. This depicts the essence of the concept elaborated by the development of MML. It can be derived directly from the MML model elements: media components represent application entities; this is modeled by MediaRepresentation relationships. User Interface components also represent application entities which is modeled by UIRepresentaion relationships. The media components can realize user interface components which is modeled by AIURealization relationships.

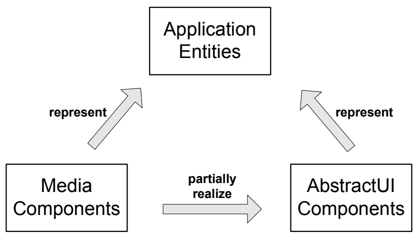

Figure 8. Interrelations between application logic, media, and user interface

7 Conclusion and Outlook

In the current paper we introduce MML a platform-independent visual modeling language for the design and model-driven development of multimedia applications. MML integrates two different research lines: on the one hand application oriented approaches, in particular our work in [7], and on the other hand user interface oriented approaches as discussed in [12].

The resulting modeling language differs from existing approaches in large parts: while the scene model stays similar to [7] (section 3), the application structure has to be adapted to enable the integrated approach (section 2). The presentation model from [7] is replaced by an abstract user interface model according to the issues discussed in [12] (section 4). Finally, we contribute an interaction model which is illustrated by a comprehensible example (section 5).

Beside the modeling language itself, the main general contribution of the paper lie in the proposal of a design process for multimedia applications (section 6). It aims to bridge the gap between requirements and analysis, which has ever been one of the core problems in multimedia application development. In particular we provide a structured method for the integration of the different design tasks – software design, user interface design, and media design – during the development. In general, the platform-independent language MML may provide a contribution for a better overall understanding of multimedia applications.

Future work will aim for the validation and refinement of the MML modeling concepts. An example is the development of more intuitive notations for some parts of the model. Examples are descriptive icons for the media components, the specification of a scene's class properties (i.e. attributes and operations) directly within the scene model, and a more explicit specification of objects in the interaction model. Such refinements will help to increase the usability of the language and its acceptance. We currently try to gain more experience with the application of MML by various student projects, e.g. the annually course "Multimedia Programmierung" (multimedia programming) where several student teams with six or seven persons each develop multimedia applications of medium size. For example in the last course they developed breakout games with extensive functionality and multiplayer support.

Regarding tool support we are currently developing a modeling tool for MML based on the *Eclipse* platform and related frameworks like the *Eclipse Modeling Framework*. We aim for code generators for various target platforms, like Flash or SVG/JavaScript. Prototypical implementations for the modeling tool and a SVG/JavaScript generator already exist. A code generator for Flash is currently under development. In addition we currently develop a plug-in for the Flash authoring tool to support the further processing of the generated code skeletons, e.g. the navigation through the generated application code according to the MML model.

References

[1] T. Arndt. The Evolving Role of Software Engineering in the Production of Multimedia Applications . In *IEEE International Conference on Multimedia Computing and Systems (ICMCS) 1999 Proceedings*. IEEE Computer Society, 1 edition, 1999.

[2] A. Bianchi, P. Bottoni, and P. Mussio. Issues in Design and Implementation of Multimedia Software Systems. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS '99), Florence, Italy, Volume I*, pages 91–96. IEEE Computer Society, 1999.

[3] P. P. da Silva and N. W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 117–132. Springer, 2000.

[4] G. Engels and S. Sauer. Object-oriented Modeling of Multimedia Applications. In S. K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 2, pages 21–53. World Scientific, Singapore, 2002.

[5] J. Gómez, C. Cachero, and O. Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia*, 8(2):26–39, 2001.

[6] A. Hannington and R. Karl. Towards a Taxonomy for Guiding Multimedia Application Development. In *9th Asia-Pacific Software Engineering Conference (APSEC 2002), 4-6 December 2002, Gold Coast, Queensland, Australia*. IEEE Computer Society, 2002.

[7] H. Hußmann and A. Pleuß. Model-Driven Development of Multimedia Applications. In *Talk at 'The Monterey Workshop 2004 - Workshop on Software Engineering Tools: Compatibility and Integration', Submitted for Proceedings*. 2004.

[8] N. Koch and A. Kraus. Towards a Common Metamodell for the Development of Web Appliactions. In J. M. C. Lovelle, B. M. G. Rodrguez, L. J. Aguilar, J. E. L. Gayo, and M. d. P. P. Ruz, editors, *Web Engineering, International Conference, ICWE 2003, Oviedo, Spain, July 14-18, 2003, Proceedings*, volume 2722 of *Lecture Notes in Computer Science*. Springer, 2003.

[9] P.-A. Muller, P. Studer, and J. Bézivin. Platform Independent Web Application Modeling. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, Lecture Notes in Computer Science. Springer, 2003.

[10] Object Management Group. UML 2.0 Superstructure Final Adopted Specification , 2004.

[11] F. Paternó, C. Mancini, and S. Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Proceedings Interact'97*. Chapman & Hall, 1997.

[12] A. Pleuß. Modeling the User Interface of Multimedia Applications. In L. Briand and C. Williams, editors, *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3731 of *Lecture Notes in Computer Science*. Springer, 2005.

[13] A. Rahardja. Multimedia Systems Design: A Software Engineering Perspective. In *International Conference on Computers and Education (ICCE) 95 Proceedings*. IEEE Computer Society, 1995.

[14] S. Sauer and G. Engels. Extending UML for Modeling of Multimedia Applications. In M. Hirakawa and P. Mussio, editors, *IEEE Symposium on Visual Languages 1999 Proceedings*. IEEE Computer Society, 1999.

[15] P. Szekely. Retrospective and Challenges for Model-Based Interface Development. In J. Vanderdonckt, editor, *Computer-Aided Design of User Interfaces*. Presses Universitaires de Namur, Namur, Belgium, 1996.

[16] H. Trætteberg. *Model-based User Interface Design*. PhD thesis, Norwegian University of Science and Technology, Oslo, 2002.

[17] J. Van den Bergh and K. Coninx. Towards Modeling Context-Sensitive Interactive Applications: the Context-Sensitive User Interface Profile (CUP). In *ACM Symposium on Software Visualization (SoftVis 2005)*, volume 1. ACM Press, 2005.