Model-driven development of interactive multimedia applications with MML

Andreas Pleuss and Heinrich Hussmann

Abstract There is an increasing demand for high-quality interactive applications which combine complex application logic with a sophisticated user interface, making use of individual media objects like graphics, animations, 3D graphics, audio or video. Their development is still challenging as it requires the integration of software design, user interface design, *and* media design.
This chapter presents a model-driven development approach which integrates these aspects. Its basis is the *Multimedia Modeling Language* (*MML*), which integrates existing modeling concepts for interactive applications and adds support for multimedia. As we show, advanced multimedia integration requires new modeling concepts not supported by existing languages yet.
MML models can be transformed into code skeletons for multiple target platforms. Moreover, we support the integration of existing professional multimedia authoring tools into the development process by generating code skeletons which can be directly processed in authoring tools. In this way the advantages of both – systematic model-driven development *and* support for creative visual design – are combined.

1 Introduction

With the evolution of end-user oriented applications in the last years – like the advancements in web applications, mobile applications, entertainment, or infotainment area – it has become widely accepted that a sophisticated user interface can significantly contribute to an application's success. Such user interfaces are often

Andreas Pleuss
Lero, University of Limerick, Ireland, e-mail: andreas.pleuss@lero.ie

Heinrich Hussmann
University of Munich, Germany, e-mail: hussmann@ifi.lmu.de

highly interactive, provide a sophisticated user interface, and – depending on the purpose – make use of multimedia capabilities. Typical reasons for multimedia usage are 1) to enhance efficiency and productiveness of the user interface, 2) to achieve more effective information and knowledge transfer, and 3) to provide enhanced entertainment value [10].

This work deals with the development of interactive multimedia applications. While the application areas mentioned above are typical for multimedia usage, multimedia user interfaces can be found in almost any application area today. Traditionally, the term multimedia has been understood as a composition of continuous (like audio, video, and animations) and discrete media elements (like 2D and 3D graphics, text, and images) into a logically coherent unit [2]. However, from the viewpoint of application development, the main difference today is much more the integration of *non-standard* media objects into the application. This requires specific experts and tools, like for graphics design, video production, or 3D design. Thus, here we understand the term "'multimedia application"' in a broad sense as *any kind of interactive application integrating individual media objects* (like graphics, animations, audio or video) *to an extent relevant for its development*.

The development of multimedia applications still lacks a systematic development approach. Traditional approaches from multimedia domain provide extensive support for media creation but neglect the application logic and Software Engineering principles [9, 15, 12]. On the other hand, existing approaches from software engineering do not support user interface and media aspects yet (see Section 2).

In our opinion, the most important differences between the development of multimedia applications and conventional application development (as considered in Software Engineering) are:

1. *Interdisciplinary development:* Multimedia application development involves three different kinds of design: 1) *Software Design*, as in conventional software development, for developing the application logic 2) *User Interface Design*, as usability is strongly important for multimedia applications, and 3) *Media Design* as creation of media objects requires usually specific knowledge and tools. Thus, different developers groups, tools, and artifacts have to be integrated into the development process.
2. *Importance of non-functional requirements:* Requirements like entertainment value, usability, and aesthetics, are strongly important for multimedia applications. Thus, visual authoring tools focusing on the creative, artistic visual design, like *Adobe Flash* or *Adobe Director*[1], have been established as development tools [3, 7].

In our work we aim to address these challenges by a model-driven development approach which integrates the different developer groups and the artifacts they produce. For this, we provide a modeling language that integrates software design, user interface design, and media design into a single, consistent language. The models hence provide a kind of contract between the different developer groups, so that all developed artifacts will fit together.

[1] http://www.adobe.com/

From the models, we then automatically generate code skeletons. As our modeling language is platform-independent, it is possible to generate code for any target platform. In particular, to integrate the existing established authoring tools, we support them as target platforms and generate code skeletons which can be directly processed within these tools. In this way, the development process becomes much more systematic while still leveraging these established tools for the final user interface and media design.

The remainder of this chapter presents our modeling language for multimedia applications. The language integrates concepts from areas of Software Engineering and model-based user interface development and extends them by new concepts required for advanced multimedia integration.

For the purpose of this chapter, we use a 2D racing game application as a running example. Gaming applications are well-suited examples because they are commonly understood and make use of both, 1) a very complex and individual user interface and 2) complex application logic. Nevertheless, it is important to note that our approach is not restricted to any specific application domain and has already been applied to many other kinds of multimedia applications (see Section 9).

Our language supports five kinds of models: The *Task Model* describes the user tasks to be supported by the application. It uses the existing *ConcurTaskTree* notation [16] and is thus not further discussed in this chapter. The other models are the *Structure Model*, the *Scene Model*, the *Presentation Model*, and the *Interaction Model*, which are explained in the following. Afterwards, we give an overview on the language, the interrelations between the different models, and the modeling process. Finally, we describe the existing tool support and the basic concepts for the code generation.

2 Related Work

This section briefly presents related approaches. As interactive multimedia applications integrate different aspects – application logic, user interface, and media – it is related to various existing modeling approaches.

To model the application logic, the *Unified Modeling Language* (*UML*) [14] can be used. However, UML on its own is not sufficient for multimedia applications as it does not cover neither the user interface aspect nor media types.

The user interface aspect is addressed by various approaches from the area of *Model-based User Interface Development* (*MBUID*) [24] (including model-driven approaches as described in this book). Their main concepts can be summarized as follows [4]: The *Task Model* specifies the user tasks supported by the application, e.g., specified as ConcurTaskTrees [16]. It is usually complemented with a *Domain Model*, which can be a conventional UML class diagram. Based on the Task Model and the Domain Model, the *Abstract User Interface Model* (*AUI*) specifies the user interface in terms of *Abstract Interaction Objects* (*AIO*s) which are platform- and

modality-independent abstractions of user interface elements. The *Concrete User Interface Model* (*CUI*) refines the AUI for a concrete target platform.

Currently, a large amount of approaches from this area exist which nowadays have evolved towards the model-based and model-driven approaches described in this book, considering advanced user interface issues like specific target devices or context-sensitivity. However, as existing approaches address user interfaces built from standard widgets, they on their own are not sufficient for interactive multimedia. However, they provide the basic concepts for the user interface aspect in MML.

The area of *Web Engineering* [11] targets model-driven development of web applications. Typical models, besides a *Domain Model*, are the *Hyperlink* or *Navigation Model* which shows the links and navigation structure of the application, and the *Presentation Model* which specifies the look and feel of the user interface and sometimes also its behavior. While earlier approaches mainly address applications with HTML-based user interfaces, latest work focuses on Rich Internet Applications [5, 25, 23]. However, they still address user interfaces made of standard widgets while individual multimedia user interface are not supported yet.

Finally, a few modeling approaches exist which already address multimedia. However, most of them [8, 26, 2] focus on multimedia documents but do not cover interactive applications. An exception is OMMMA [6] which supports interactive multimedia applications as considered in this chapter. However, OMMMA does not integrate the results from MBUID area and also lacks of the advanced concepts we discuss in Section 3. Nevertheless, it provides substantial basic concepts which have been included into MML.

First concepts of MML have been presented in [18, 17]. However, as discussed in [20], there is a need for advanced modeling concepts for Media Components, like different abstraction layers, inner structure, and variations (see Section 3). Moreover, the language has evolved based on the experience from its usage in several student projects (see Section 9). In this chapter we present the resulting integrated version of MML. A full language reference can be found in [19].

3 MML Structure Model

The MML Structure Model describes the structure of the application. Fig. 1 shows an example of a racing game application and is used throughout this section to illustrate the introduced concepts.

The Structure Model contains the Domain Classes for the application logic. They are described like in a conventional UML class diagram. For instance, a racing game might contain classes Race, Car, Player, and Track. A Track contains Obstacles and Checkpoints (like the start and the goal). Domain Classes have properties and relationships like in conventional UML class diagrams (Fig. 1).

In addition, the media elements are basic assets of the application as well and their production can require much effort, specific experts, and specific tools. In addition, the usage of a specific kind of media content can be an essential requirement

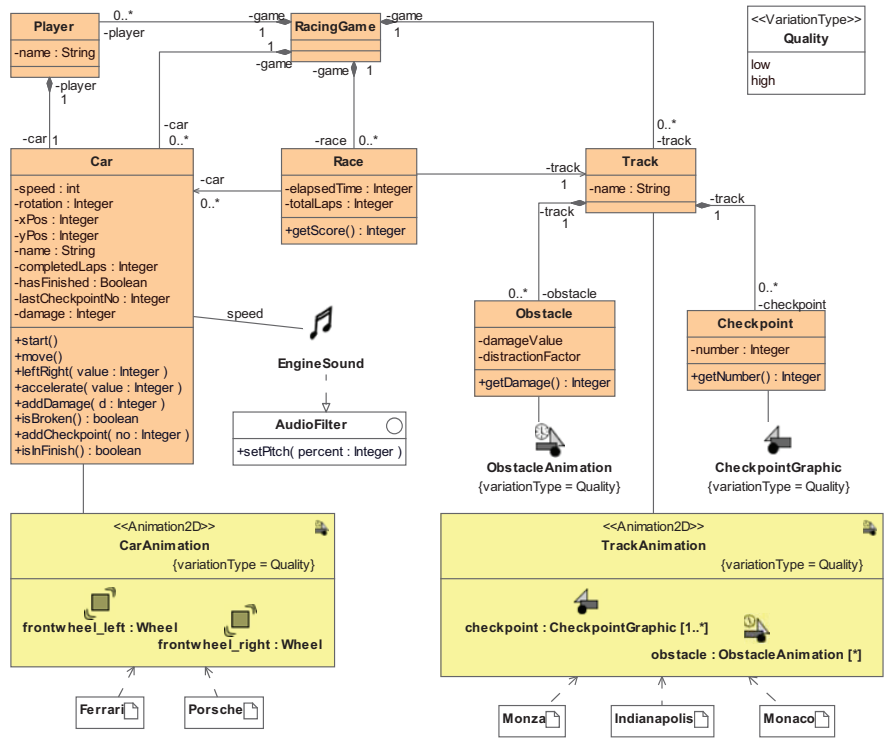

Fig. 1 MML Structure Diagram Example

for the application. For instance, the customer might want the racing game application to use 3D graphics or an e-learning application to contain videos. For these reasons, the media elements are modeled in MML as first-class entities in the Structure Model.

In interactive applications, media content is often associated with some functionality to render and control the media content. For instance, a video is usually shown in a video player which allows to play, stop, rewind, etc., the video. Thus, in MML the media content is encapsulated together with basic playing and rendering functionality as a *Media Component*. Each Media Component is of a certain media type which an be *Audio*, *Video*, *2D Animation*[2], *3D Animation*, *Graphics* (i.e., vector graphics), *Image* (i.e., raster graphics), or *Text*.

A Media Component is denoted in MML similar to a Component in UML as a rectangle with several optional compartments for additional properties. Like UML, MML allows to choose between different alternative notations. The media type is denoted by an icon and/or a keyword. In Fig. 1, the Media Components CarAnimation and TrackAnimation are displayed with a compartment showing their inner structure

[2] "Animation" here refers to any kind of change over the time, like changing its position on the screen or changing its shape

(explained below). The Media Components EngineSound, ObstacleAnimation, and CheckpointAnimation are denoted in collapsed notation with an icon only.

Media Component are associated with Domain Classes which they represent. This is specified in MML by a *Media Representation* relationship between a class and a Media Component. In the example, the Domain Class Car is represented by CarAnimation and EngineSound. Obstacle and Track are represented by animations as well (ObstacleAnimation and TrackAnimation) while Checkpoint is represented by static graphic (CheckpointGraphic). To specify that the Media Component represents a specific property or operation of the class, Media Representation relationship is marked with the name of the property or operation. For example, EngineSound represents the property speed of Car.

MML defines some standard operations for each media type, which have not to be modeled explicitly and can be used in the Interaction Model (see Section 6) to specify a Media Component's behavior. For instance, a 2D Animation provides operations to access its coordinates on the screen and its size. It is also possible to define custom operations for Media Components in terms of Interfaces (like UML Interfaces) which can be associated with Media Components. An example is the interface AudioFilter which is associated with EngineSound in Fig. 1.

3.1 Abstraction Layers for Media Components

In [20] we have discussed advanced concepts for modeling Media Components in interactive applications that have not been addressed so far. An important finding is that different abstraction layers have to be considered: Media Components, like CarAnimation, are abstract constructs which might be realized by multiple concrete artifacts, like Porsche and Ferrari. It is beneficial to consider them as abstract elements, instead of dealing with the concrete artifacts only, because in this way all kind of cars can be handled in the same way. Moreover, the concrete artifacts are often unknown in early phases, because it is not decided yet how many different cars the final application will provide. In fact, in some cases the concrete artifacts are not known at all at the design time, because they are dynamically loaded from a server or because the user creates them himself dynamically at runtime (e.g., using a "car editor" delivered with the application). Even when the concrete artifacts are known, it might be desired not to model all of them, because their number is too large and they will be loaded from a database later.

On the other hand, if the concrete artifacts are already known, there should be a way to specify them in the model. Thus, we introduce *Media Artifacts* which can be used to optionally specify concrete artifacts. In the example, CarAnimation is manifested by two Media Artifacts Porsche and Ferrari and TrackAnimation by three Media Artifacts Monza, Indianapolis, and Monaco.

One should note that there is a third abstraction layer for Media Components which are the concrete instances of Media Components on the user interface (see Section 5). Of course, a Media Component can be instantiated multiple times, like in

a racing game where usually multiple cars take part. Again, it is possible to specify the concrete Media Artifact used for a Media Instance but not mandatory; sometimes they might be decided at design time (e.g., in the first level of the racing game the user has always to use the Porsche) while sometimes it might be decided at runtime (e.g., the user can select the car herself).

3.2 Inner Structure of Media Components

A second important finding from [20] is the need to define the inner structure of Media Components. This is in particular important for interactive multimedia applications: For instance, let us consider, that the CarAnimation's front wheels should turn when the car drives through a turn. Then, the media designer needs to know that the car's front wheels have to be designed as own (graphical) objects which can be accessed and modified by the application logic. The software designer in turn needs to know how to access them (e.g., the names assigned by the media designer). Thus, it is necessary to specify such inner structure in the model as a kind of contract between the developers.

It is important to note that it is not intended in MML to model the complete (visual) structure of the car. The inner structure is modeled only when it should be accessed by application logic. This happens either when some media parts should be accessed or modified, like in the example above, or when an event listener should be attached to a media part (like, for instance, that the user can trigger some action by clicking on the car's wheels).

Again, the different abstraction layers have to be considered in a consistent way. In MML, a *Media Part* represents an (abstract) part of a Media Component, like *Wheel*. Each Media Part has a type depending on the Media Component it belongs to. For instance, a video can consist of *Audio Channels* and *Image Regions* while a 3D animation consists of *3D Objects*, *Transformations*, *Light*, etc. 2D animations, like in the racing game example, consist just of graphical objects which we call *SubAnimations*.

A Media Part can be instantiated multiple times, acting in multiple roles, similar like properties in a conventional UML class. For instance, a car has multiple wheels like frontwheel_left and frontwheel_right[3]. These instances are called *Inner Properties* (to distinguish them from other properties of the Media Component).

An Inner Property can also be an instance of another Media Component, like in TrackAnimation, which contains multiple instances of ObstacleAnimation and CheckpointGraphic. As shown in the example, a multiplicity can be specified for each Inner Property. The software developer can then access the single instances in a way similar to arrays, e.g., 'wheel[1]'.

Analogously to Media Components, it is optionally possible to specify a concrete artifact for a Media Part, called *Part Artifact*, which can be useful if the developer

[3] Note that the back wheels have not to be modeled here as they need not to be accessed by application logic

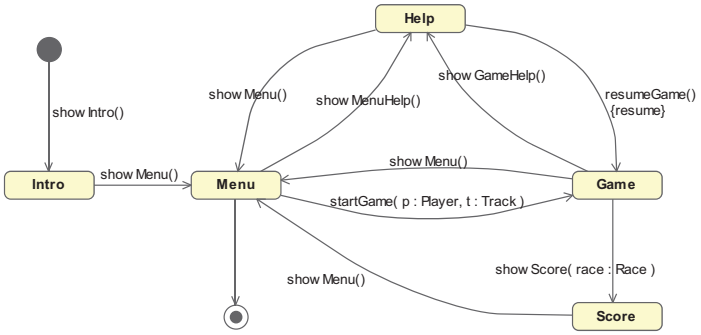

Fig. 2 MML Scene Diagram Example

wants to distinguish explicitly between PorscheWheel and FerrariWheel or to express that all car types reuse the same artifact for their wheels (specified by marking an Inner Property with the keyword unique) (see [20] for in-depth discussion of all possible cases).

3.3 Variations of Media Components

A third concept, not discussed in previous work so far, is the need for an efficient way to specify different variations of Media Components. For instance, Media Components frequently have to be provided, e.g., in different qualities or different languages. Therefore, in MML the modeler can introduce in MML a *Variation Type* (like *Quality* in Fig. 1) and specify different *Variation Literals* for it, like high and low. This means that each Media Component, where the Variation Type is assigned to, must be created in each possible variation. For instance, all visual Media Components in the example have assigned the Variation Type Quality which means that the media designers have to provide them in the two different variations high and low. Basically, the variations could also be combined with mechanisms for context-sensitive user interfaces from MBUID (see Section 2), e.g., to select the appropriate variation automatically at runtime based on the application context.

4 MML Scene Model

The Scene Model describes the application's coarse-grained behavior or navigation in terms of *Scenes*. A Scene represents an application state associated with a corresponding user interface. For instance, in the racing game, the scenes are Intro, Menu, Help, Game, and Score. The Scene model shows the Scenes and the transitions between them using an adapted notation of UML State Charts (Fig. 2).

An important multimedia-specific aspect of Scenes is their dynamic character. One the one hand, this is caused by dynamic behavior of time-dependent media instances (audio, video, animations) in the Scenes. On the other hand, user interfaces are often generated dynamically at runtime, like the number of cars taking part in a racing game or the track chosen by the user. Thus, Scenes can be generic and receive parameter values. Therefore, each Scene has *Entry Operations* and *Exit Operations*. Entry Operations are used to initialize the Scene and pass parameters to it. Exit Operations are used to clean up a Scene and invoke another Scene.

As shown in Fig. 2 for the racing game example, the Scenes are denoted as states with the possible transitions between them. The transitions are annotated with the names of Entry Operations executed in the target Scene when performing the transition. Exit Operations need not to be modeled explicitly as their names are by convention derived from the transitions in the diagram.[4].

By default, executing an Entry Operation initializes a Scene. However, sometimes a Scenes has already been active before and its previous state should be resumed. This is specified by attaching the keyword resume to the Entry Operation. For instance, when the user calls the Help during the Game he/she probably wants to resume the game after the consulting the help. Thus, the Entry Operation resumeGame() is marked with this keyword (Fig. 2).

Beside Entry and Exit Operations it is also possible to define additional properties and operations for a Scene. Moreover, Media Components can not only represent Domain Classes, as explained for the Structure Model, but also Scenes. For instance, a Media Component HelpText would probably not be associated with one of the Domain Classes but with the Scene Help. Analogously to the Structure Model, this is specified by a Media Representation relationship between the Media Component and the Scene.

5 MML Presentation Model

The MML Presentation Model specifies the user interface for each Scene. It is initially modeled using Abstract Interaction Objects as common in the MBUID area (see Section 2). However, as we deal with multimedia applications, in a second step the instances of the Media Components from the Structure Model come into play.

5.1 Abstract User Interface

In MML, each Scene is associated with a *Presentation Unit*. A Presentation Unit is an abstraction from a screen in a graphical user interface. It contains *Abstract Interaction Objects* (*AIO*) which are platform- and modality-independent abstractions of

[4] the names are composed of a prefix 'exitTo', the name of the target scene, and the name of the target Entry Operation, separated by _, e.g. exitTo_Menu_showMenu()

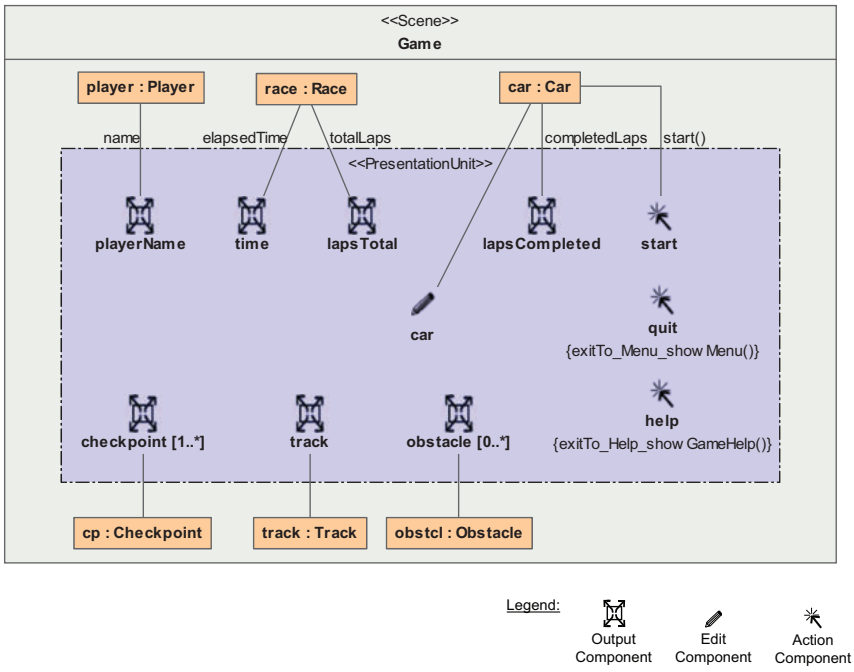

Fig. 3 AIOs in the MML Presentation Model for the Scene Game.

user interface elements. For instance, an *InputComponent* enables the user to input some data, an *Output Component* presents some data to the user, an *Edit Component* combines input and Output Component, and an *Action Component* allows the user to trigger an action. It is also possible to apply further concepts from MBUID here, like modeling the layout of the AIOs, but this is not further discussed here.

Fig. 3 shows as an example the Presentation Unit for the Scene Game. The Scene acts as the overall container. It contains *Domain Objects*, which are instances of the Domain Classes from the Structure Model, and a Presentation Unit containing the AIOs. The Presentation Unit in the Scene Game contains, for instance, several Output Components to show the track, the obstacles, and the checkpoints. The player's car is represented by an Edit Component as it presents the current state of the car which is also manipulated by the user. There are some Output Components for additional information, like the playerName, time, and lapsCompleted, and some Action Components to start the race or to navigate to the help or back to the menu.

Each AIO represents a Domain Object as specified by *UI Representation* relationships. Analogous to Media Representation relationships in the Structure Model, it is possible to annotate the name of a property or operation to specify that this specific property or operation is represented by the AIO. For instance, the Output Component playerName represents the property Name of player. AIOs not associated with a Domain Object represent the Scene itself. In this case, the name of a Scene's property or operation represented by the AIO is denoted directly below the AIO (in

curly braces), like for the Action Component quit which triggers the Scene's Exit Operation exitTo_Menu_showMenu() (see Section 4).

Due to the possible dynamic character of multimedia user interfaces, it is possible to specify a multiplicity for an AIO (like for obstacle) as their number is calculated dynamically at runtime. Also, the Interaction Model can be used to specify that AIOs are added or removed from the user interface. To indicate that an AIO is initially invisible on the user interface and becomes visible at runtime, the keyword invisible can be attached to the AIO.

5.2 Multimedia User Interface

As we deal with multimedia applications, some of the AIOs can be realized by media objects. Therefore, the respective AIO is connected with a *Media Instance* by a *UI Realization* relationship. Its semantics is that the AIO is implemented by the Media Instance. In turn the Media Instance must provide the interaction concepts defined by the AIO. For instance, if an animation should realize an Action Component, this means that, e.g., clicking on the animation triggers some action. Thus, not any type of Media Instance can realize any AIO (see discussion in [18]). AIOs which are not realized by a Media Instance are intended to be implemented in conventional way, i.e. using standard widgets.

Media Instances are instances of Media Components from the Structure Model. So, a Media Component can be used in different Scenes. For instance, the CarAnimation is not only used in the game itself but also in the menu where the user can select between different cars.

Fig. 4 shows the Presentation Model for the Scene Game enhanced with Media Components. For instance, the AIO car is realized by instances of the Media Components CarAnimation and EngineSound. It is also possible that AIOs are realized by inner objects of a Media Instance, like checkpoint and obstacle in the example.

Like in existing MBUID approaches, the AIOs trigger events during the user interaction; e.g., an Action Component triggers an operation. However, Media Instances of temporal media type can trigger additional events independent of the user. For instance, an audio object can trigger an event when it finishes playing or a moving animation can trigger an event when collides with another one. This is modeled in MML using the concept of *Sensors* adapted from 3D graphics domain [27]. In Fig. 4, two Collision Sensors (checkpointSensor and obstacleSensor) are associated with the CarAnimation instance. They test whether the car animation collides with checkpoints or obstacles and trigger an event when this occurs.

Another type of sensors is the *Visibility Sensor* which triggers an event when an object becomes visible, e.g. after it has been covered by another object or was located outside the screen. A *Proximity Sensor* is relevant for 3D objects only and triggers an event when the user navigates within a 3D world close to this object. A *Time Sensor* triggers an event at specific points of time in the application. In MML

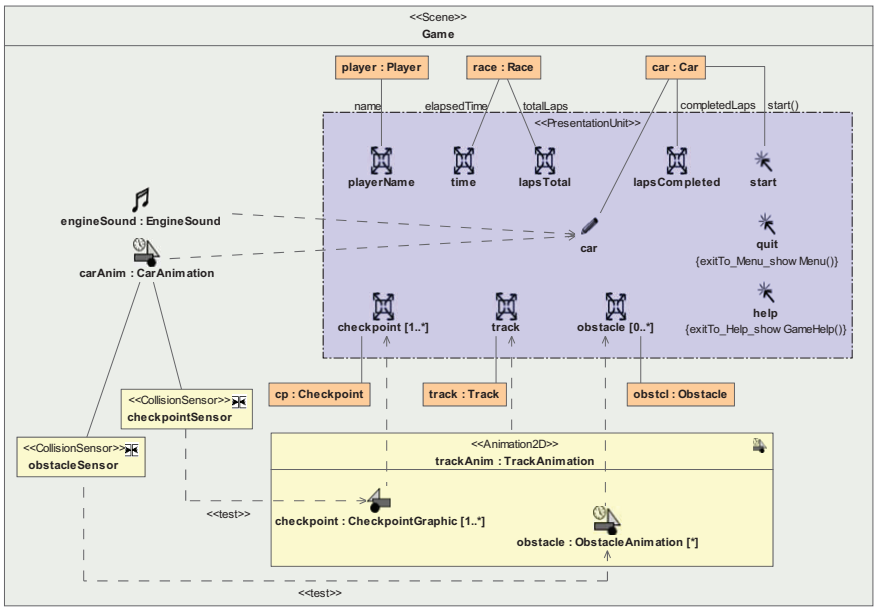

Fig. 4 MML Presentation Diagram enhanced with Media Instances and Sensors for the Scene Game

this points in time can be either defined by fixed time interval or by one or more *Cue Points* of temporal Media Instances.

A *Cue Point* (not shown in the example) can be defined for any temporal media object and allows to refer to a specific point in time on the media object's time line. This can be used to specify synchronization between temporal media objects. Let us consider that the racing game application shows as introduction a video and some animated text. The text should appear after the first scene in the video is finished. Therefore, a Cue Point can be defined, like "firstSceneFinished". Using a Time Sensor it is now possible to specify that, when firstSceneFinished occurs, an operation is triggered on the Media Component for the text, like setVisible(). The advantage of Cue Points compared to concrete time values is their abstract character as often the concrete duration of the video is still unknown or may change later according to the aesthetic considerations of the media designer.

6 MML Interaction Model

The Interaction Model specifies the user interaction and the resulting behavior of the Scene. The core idea is to specify how events initiated by the user interface trigger operations of Domain Objects or of user interface elements (AIOs or Media Components). In that way the Interaction Model specifies the interplay between the

elements defined in the foregoing models. The behavior of Domain Class operations itself is not part of the MML models as it is often quite complex and usually specified directly in a programming language – in particular, as the operations in a multimedia application (like moving the car in a realistic way) often require much trial and error and cannot be specified in advance.

Existing work in MBUID often uses Task Models to describe the interaction. As mentioned in the Section 1, MML supports Task Models as well. They are sufficient to specify the interaction for less dynamic Scenes, like the Menu in the racing game. However, dynamic multimedia Scenes, like the Scene Game, often require a more detailed modeling of temporal behavior. Therefore, such Scenes are modeled using the MML Interaction Model which is an adapted UML Activity Diagram. As shown in existing work [1] extended UML Activity Diagrams can also be used to specify similar operations like in Task Models, while on the other hand, as sometimes used in UML, they also allow to model very detailed object-oriented behavior.

The objects which can be used in a Scene's Interaction Model are the Domain Objects, AIOs, Media Instances, and Sensors owned by the Scene as defined in the Scene's Presentation Model. Additional objects can be passed as parameters of the Scene's Entry Operations. An *Actions* in the Interaction Model refers to an operation call (like UML *CallOperationActions*) on one of the objects owned by the Scene. In this way, by restricting the Activity Diagram to defined objects and operation calls, it is possible to directly generate code from the model.

As mentioned before (Section Section 3), MML defines some standard operations for Media Components, like start, stop, play() for videos. Analogously, some standard operations are predefined by MML for the AIOs (e.g., disable(), setVisible()) to save the modeler defining them manually.

Beside the operation calls, the Interaction Model contains events triggered by AIOs and Sensors to model the interaction. They can be used analogous to *AcceptEventActions* in UML, e.g., in combination with *InterruptibleActivityRegions* whose execution terminates when they are left via an interruptible edge. In this way, it can be specified that, e.g., some user input interrupts the current program flow (i.e. terminates tokens) and starts another one.

Fig. 5 shows an simplified extract from the Interaction Model for the Scene Game. Like in UML Activity Diagrams, Actions have *Input Pins* to specify the parameters for an operation. An Input Pin called target is used to specify the object on which the operation call is executed. Input Pins marked with the name of an operation argument receive an argument of the operation, like value for leftRight(). In the simplified model, the input value to control the car is received from the Edit Component car, and passed as parameter to the operation leftRight() of the Domain Object car. Afterwards, the operation move() is called on car. This is executed in a loop until an interruption occurs by the collision sensor obstacleSensor or by the ActionComponent quit.

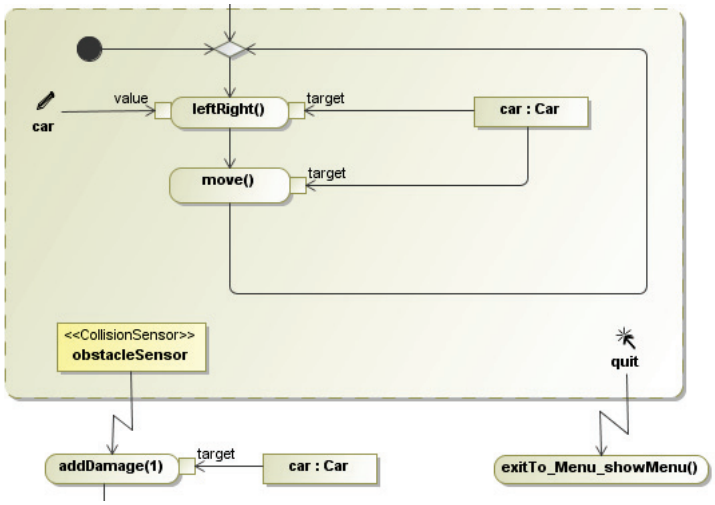

Fig. 5 Simplified extract from the Interaction Model for the Scene Game

7 Overall Approach

Fig. 6 shows the overall modeling process with MML. The horizontal axis shows the different developer roles involved in the process, i.e. software design, user interface design, and media design. The vertical axis represents the temporal dimension. The center shows the different MML models and the interrelations between them.

The modeling process is performed during the design phase of the application. It starts after the requirements analysis, which is performed in the usual way and not further considered here. The Task Model reflects the user tasks to be supported by the application from the viewpoint of user interface design. They have to be derived from the requirements specification. In parallel, the Structure Model is specified which consists of Domain Classes and Media Components. Domain Classes can be derived from the requirements specification similar to conventional object-oriented development.

The Media Components are derived from the requirements specification as well, as far as they are specified therein. Domain Classes and Media Components are related through Media Representation relationships. Thus, adding Media Components can require additional Domain Classes which represent associated application logic. In turn, for each Domain Class can be considered whether it is useful to represent it by a Media Component. The Structure Model should be created in cooperation between Software Designer and Media Designer as it defines how application logic can access Media Components and how those must be structured for this purpose.

The Scene Model describes the Scenes and the navigation between them. The decomposition into Scenes influences the application's usability and is thus specified by the user interface designer. The Scenes can be identified based on the Task Model, for instance using an approach like in [13].

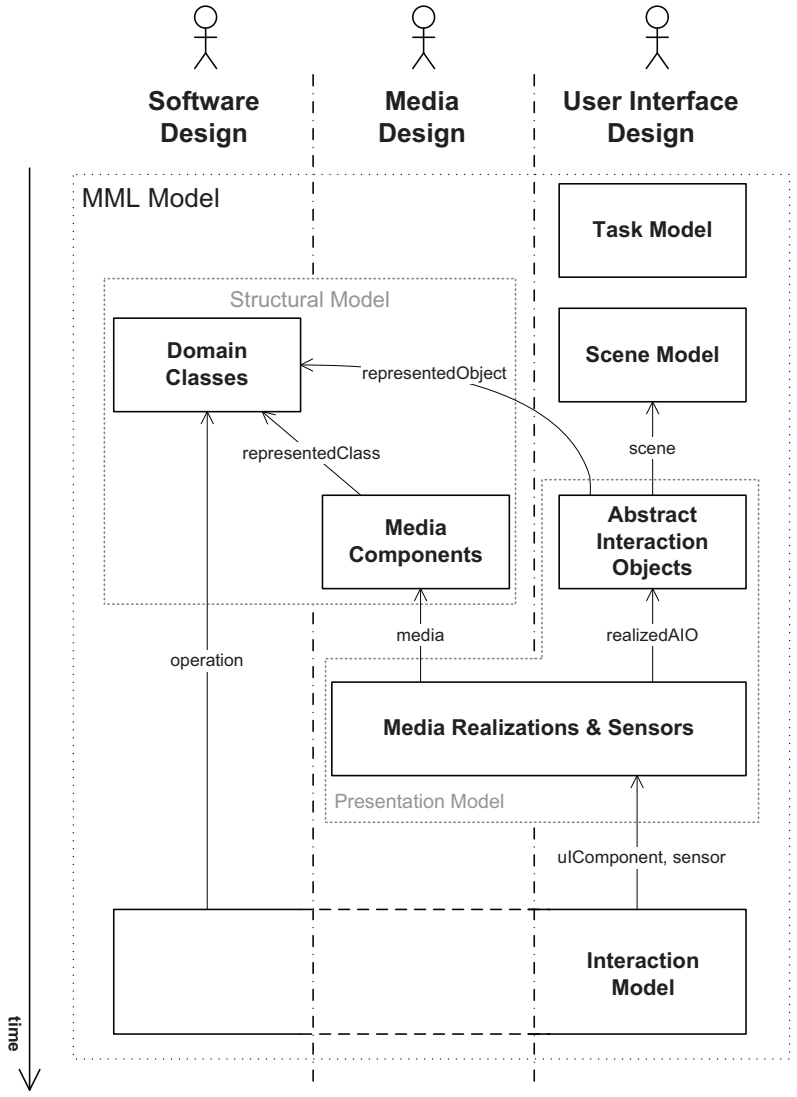

Fig. 6 MML overall modeling process

For each Scene, a Presentation Model is defined by the user interface designer. It specifies the AIOs which can be derived from the Task Model as well. The AIOs are associated with Domain Objects which are instances from the Domain Classes in the Structure Model. In the next step, the Presentation Model is complemented with Media Components and Sensors. At this point, the user interface designer and the media designer have to cooperate. The Media Instances refer to Media Components in the Structural Model. During these steps, missing Domain Classes and missing

Media Components can be identified and added. In addition, the user interface designer and the media designer can add Sensors to Media Components. Basically, it is also possible to refine the Presentation Model in terms of a Concrete User Interface Model as in existing MBUID approaches.

Finally, the Interaction Model is specified in cooperation between the software designer and the user interface designer. It specifies how user interface events and Sensor events trigger operation calls on domain objects. Missing model elements, like missing objects in the Scene or missing class operations, can be identified and added. The user interface designer is responsible for the interaction which can – at least to some degree – be derived from the Task Model. The software designer's knowledge is mainly required for specifying the more complex behavior of dynamic Scenes like the Game Scene in the racing game.

It is not mandatory to follow the modeling process always as described here. Basically it is possible to start with any kind of MML model or to specify the model in several iterations. For example, it is possible to start the process with the Scene and Presentation Models and to create the Structure Model on that base. Indeed, it is also possible that all three developer groups iteratively specify all models in cooperation or that there is an additional modeling expert who supports the different developers in specifying the models.

8 Tool Support and Code Generation

MML is defined as a standard-compliant metamodel implemented with the *Eclipse Modeling Framework* (*EMF*)[5]. The advantage of EMF is its integration with many other existing tools from model-driven engineering, like tools for model transformation, validation, or model weaving[6]. As visual modeling tool for MML we provide an extension of the UML tool *Magic Draw*[7]. In addition, we provide a model transformation to transform the models from Magic Draw into EMF so that they can be further processed by EMF-based tools. The advantage of using Magic Draw instead of creating our own EMF-based visual modeling tool is that a professional modeling tool provides a degree of usability and robustness which is difficult to achieve with an implementation of our own.

After MML models have been transformed into the EMF-based format it is possible to generate code skeletons for different target platforms. To this end, we provide several model transformations in the *Atlas Transformation Language* (*ATL*)[8]. The currently most mature transformation is one generating code skeletons for the target platform Flash. A Flash application consists of *Flash documents* and associated code in the object-oriented programming language *ActionScript* which is part of Flash.

[5] http://www.eclipse.org/emf/

[6] http://www.eclipse.org/modeling/

[7] http://www.magicdraw.com

[8] http://www.eclipse.org/m2m/atl/

Thereby, we generate the Flash/ActionScript skeletons in such a way that they can be directly loaded into the Flash authoring tool. The designers then can process them using all the professional functionality of the tool. Other target platforms currently supported by prototypical model transformations are Java, SVG/JavaScript, and Flash Lite for mobile devices.

The basic concepts for the Flash/ActionScript code generation are as follows: The Domain Classes from the Structural Model are mapped to ActionScript classes, analogous to existing mappings from UML class diagrams to Java code. The Media Components are mapped to Flash documents containing placeholders according to the Media Component's inner structure defined in the model. These placeholders then have to be filled out by the media designers using the Flash authoring tool. In addition, an ActionScript class is created for each Media Component providing predefined operations for the Media Component and skeletons for operations defined in custom interfaces (see Section 3).

Each Scene is mapped to an ActionScript class containing the Entry and Exit Operations defined in the model. The Exit Operations contain the code for the navigation between Scenes as defined by the transitions in the Scene Model. Additionally, each Scene is mapped to a Flash document containing the Scene's user interface. For the user interface, each AIO is mapped to a corresponding widget in the Flash document and an associated ActionScript class containing event listeners. For Media Instances, an instance of the generated Media Component is placed on the user interface. All instances on the user interface have a name by which they can be accessed from the code in the Scene's ActionScript class. Sensors are implemented by corresponding code in the Scene's ActionScript class, e.g. a Collision Sensor is implemented by an operation which tests whether two visual objects overlap each other on the screen. The Interaction Model is mapped to corresponding ActionScript code in the Scene.

It is possible to directly open the generated code skeletons in the Flash authoring tool. Fig. 7 shows a screenshot of the generated skeleton for the scene Game in the Flash authoring tool. It shows the user interface elements and the placeholders for Media Components (represented by simple rectangles) generated according to the MML model from Fig. 4. The application can also be directly executed to test the navigation between the generated Scenes.

To finalize the application, the developers have to complete the ActionScript code, mainly the bodies of the Domain Class operations, as those are not specified in MML. The media designers need to fill out the placeholders generated for the Media Components using the whole functionality of the authoring tool. The generated user interface can also be freely edited, arranged, and adapted in the tool using all its visual functionality (e.g., in Fig. 7 a button is resized). However, all relationships between the different application parts – like between an AIO and its Domain Object, its event handling code, and its Scene – are generated from the model. Moreover, the generated code is consistent and well-structured (using e.g. design patterns, see [21, 19]). In this way, the advantages of both – 1) visual design of media and user interfaces using established authoring tools and 2) systematic development of well-structured application using model – are combined.

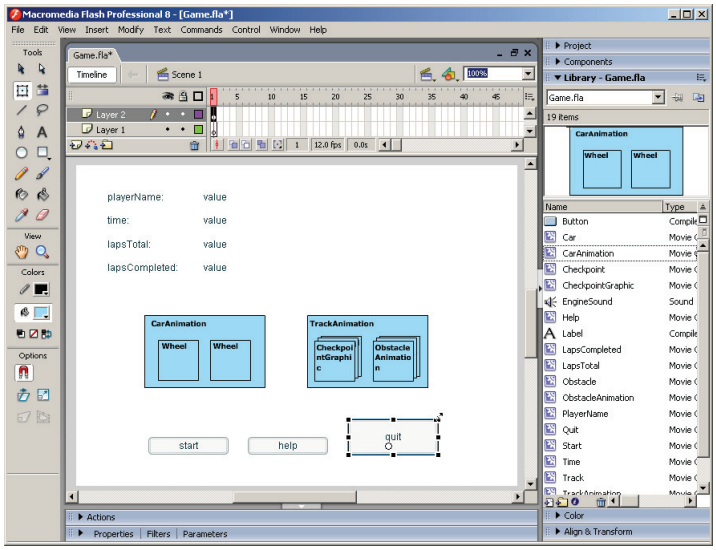

Fig. 7 Generated Skeletons for the Racing Game

9 Conclusion and Outlook

In this chapter we presented MML, a modeling language for interactive multimedia applications, and an associated model-driven development approach. It targets the needs for a systematic development process considering the characteristics of interactive multimedia user interfaces.

With respect to the main challenges for multimedia development from Section 1, we address the interdisciplinary development by integrating software design, user interface design, *and* media design into a single consistent modeling language. The MML models thus act as a kind of contract between the different developer groups. As discussed in Section 2, to our knowledge, none of the existing modeling languages covers all of these three aspects so far.

In particular, we show that modeling interactive multimedia requires new concepts like different abstraction layers (Media Components, Media Artifacts, and Media Instances) and modeling the (abstract) inner structure of Media Components. MML also demonstrates how to integrate these concepts into a consistent modeling approach. However, the general multimedia-specific concepts can also be used to extend other existing modeling approaches with multimedia support.

The need for visual, artistic design is addressed by integrating authoring tools through generation of code skeletons which can be directly loaded and processed within established professional authoring tools like Flash. This concept has been generalized beyond multimedia and Flash [22] and is an important aspect in practice as mature visual tool support is usually strongly important for user interface designers and media designers.

To validate the language as far as possible, it has been applied three times in a practical graduate course where students develop interactive Flash/ActionScript applications over three months in teams of four to seven students each. The developed applications were multi-player blockout games, multi-player jump and run games, and multi-player minigolf games. In addition, there were three student projects were a student developed a small or medium-size Flash/ActionScript application for a third party customer using MML. These were 1) an interactive multimedia application for a hairdresser 2) an interactive visual help system for a professional customer relationship management system, and 3) an authoring tool for creating commercial interactive learning systems. In these cases, the final implementations made extensive usage of the code generated from the language. Some other kinds of applications have been created and implemented prototypically using MML, including a simple navigation system and a media player application. Although this validation can not provide quantitative evidence, it strongly contributed to revisions which lead to the version of the language as presented here.

For future work, it seems beneficial to combine the elaborated concepts with those from other modeling approaches, like mobile applications, context-sensitivity for multimedia user interfaces in ambient environments, or concepts from Web Engineering for Rich Internet Applications.

References

1. Van den Bergh, J.: High-level user interface models for model-driven design of context-sensitive user interfaces. Ph.d. thesis, Hasselt University, Diepenbeek, Belgium (2006)
2. Boll, S.: Zyx – towards flexible multimedia document models for reuse and adaptation. Phd, Vienna University of Technology, Vienna, Austria (2001)
3. Bulterman, D.C.A., Hardman, L.: Structured multimedia authoring. ACM Trans. Multimedia Comput. Commun. Appl. **1**(1), 89–109 (2005). DOI http://doi.acm.org/10.1145/1047936.1047943
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., Vanderdonckt, J.: Plasticity of user interfaces: A revised reference framework. In: C. Pribeanu, J. Vanderdonckt (eds.) TAMODIA, pp. 127–134. INFOREC Publishing House Bucharest (2002)
5. Carughi, G.T., Comai, S., Bozzon, A., Fraternali, P.: Modeling distributed events in data-intensive rich internet applications. In: B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, C. Godart (eds.) WISE, *Lecture Notes in Computer Science*, vol. 4831, pp. 593–602. Springer (2007)
6. Engels, G., Sauer, S.: Object-oriented Modeling of Multimedia Applications. In: S.K. Chang (ed.) Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 21–53. World Scientific, Singapore (2002)
7. Hannington, A., Reed, K.: Factors in multimedia project and process management–australian survey findings. In: ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference, pp. 379–388. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/ASWEC.2007.22
8. Hardman, L., Worring, M., Bulterman, D.C.A.: Integrating the amsterdam hypermedia model with the standard reference model for intelligent multimedia presentation systems. Comput. Stand. Interfaces **18**(6-7), 497–507 (1997). DOI http://dx.doi.org/10.1016/S0920-5489(97)00014-7

9. Hirakawa, M.: Do software engineers like multimedia? In: Multimedia Computing and Systems, 1999. IEEE International Conference on, vol. 1, pp. 85–90vol.1 (1999). DOI 10.1109/MMCS.1999.779125
10. Hoogeveen, M.: Towards a theory of the effectiveness of multimedia systems. International Journal of Human Computer Interaction **9**(2), 151–168 (1997)
11. Kappel, G., Pröll, B., Reich, S., Retschitzegger, W. (eds.): Web Engineering - The Discipline of Systematic Development of Web Applications. John Wiley & Sons (2006)
12. Lang, M., Fitzgerald, B.: New branches, old roots: A study of methods and techniques in web/hypermedia systems design. Information Systems Management **23**(3), 62–74 (2006). DOI 10.1201/1078.10580530/46108.23.3.20060601/93708.7
13. Luyten, K.: Dynamic user interface generation for mobile and embedded systems with model-based user interface development. Ph.d. thesis, Transnationale Universiteit Limburg, Diepenbeek, Belgium (2004)
14. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.2 (2009). Formal/2009-02-02
15. Osswald, K.: Konzeptmanagement - Interaktive Medien - Interdisziplinäre Projekte. Springer, Berlin, Germany (2003)
16. Paternò, F.: Model-Based Design and Evaluation of Interactive Applications. Springer-Verlag, London, UK (1999)
17. Pleuß, A.: MML: A language for modeling interactive multimedia applications. In: ISM, pp. 465–473. IEEE Computer Society (2005)
18. Pleuß, A.: Modeling the user interface of multimedia applications. In: L.C. Briand, C. Williams (eds.) MoDELS, *Lecture Notes in Computer Science*, vol. 3713, pp. 676–690. Springer (2005)
19. Pleuss, A.: Model-driven development of interactive multimedia applications. Ph.D. thesis, University of Munich, Munich, Germany (2009)
20. Pleuss, A., Botterweck, G., Hussmann, H.: Modeling advanced concepts of interactive multimedia applications. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing VL/HCC 2009, pp. 31–38 (2009). DOI 10.1109/VLHCC.2009.5295305
21. Pleuß, A., Hußmann, H.: Integrating authoring tools into model-driven development of interactive multimedia applications. In: J.A. Jacko (ed.) HCI (1), *Lecture Notes in Computer Science*, vol. 4550, pp. 1168–1177. Springer (2007)
22. Pleuss, A., Vitzthum, A., Hussmann, H.: Integrating heterogeneous tools into model-centric development of interactive applications. In: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (eds.) MoDELS, *Lecture Notes in Computer Science*, vol. 4735, pp. 241–255. Springer (2007)
23. Preciado, J., Linaje, M., Morales-Chaparro, R., Sanchez-Figueroa, F., Zhang, G., Kroiss, C., Koch, N.: Designing rich internet applications combining uwe and rux-method. In: Proc. Eighth International Conference on Web Engineering ICWE '08, pp. 148–154 (2008). DOI 10.1109/ICWE.2008.26
24. Szekely, P.A.: Retrospective and challenges for model-based interface development. In: F. Bodart, J. Vanderdonckt (eds.) DSV-IS, pp. 1–27. Springer (1996)
25. Urbieta, M., Urbieta, M., Rossi, G., Ginzburg, J., Schwabe, D.: Designing the interface of rich internet applications. In: G. Rossi (ed.) Proc. Latin American Web Congress LA-WEB 2007, pp. 144–153 (2007). DOI 10.1109/LAWEB.2007.4383169
26. Villard, L., Roisin, C., Layaïda, N.: An xml-based multimedia document processing model for content adaptation. In: P.R. King, E.V. Munson (eds.) Digital Documents: Systems and Principles, 8th International Conference on Digital Documents and Electronic Publishing, DDEP 2000, 5th International Workshop on the Principles of Digital Document Processing, PODDP 2000, Munich, Germany, September 13-15, 2000, Revised Papers, *Lecture Notes in Computer Science*, vol. 2023, pp. 104–119. Springer (2000)
27. Vitzthum, A.: Entwicklungsunterstützung für interaktive 3d-anwendungen. Ph.D. thesis, University of Munich, Munich, Germany (2008)