

Integrating Authoring Tools into Model-Driven Development of Interactive Multimedia Applications

Andreas Pleuß, Heinrich Hußmann

Department of Computer Science, University of Munich
Munich, Germany

{pleuss, hussmann}@cip.ifi.lmu.de
<http://www.medien.ifi.lmu.de>

Abstract. The *Multimedia Modeling Language (MML)* is a platform-independent modeling language for model-driven development of interactive multimedia applications. Using models provides several advantages like well-structured applications and better coordination of the different developer groups involved in the development process. However, the creative tasks – like graphical design of the user interface and the design of media objects – are better supported by traditional informal methods and tools. In particular multimedia authoring tools such as *Adobe Flash* are well established for multimedia application development. In this paper we show how MML and authoring tools can be integrated by the example of Flash. Therefore we transform the MML models into code skeletons which can be directly loaded into the Flash authoring tool to perform the creative design tasks and finalize the application. In that way, the strengths of models and authoring tools are combined. The paper shows the required level of abstraction for the models, introduces a metamodel and a suitable code structure for the Flash platform, and finally presents the transformation.

1 Introduction

Through the growing pervasion of every day life with computers, many application areas appear where rich and comfortable and eventually entertaining user interfaces become more and more natural. In this paper we deal with “multimedia user interfaces” which make intensive use of different kind of media – like audio, video, graphics, and animation – and provide sophisticated user interfaces adapted individually to the user’s tasks and information. In particular, we address highly interactive systems which may include complex application logic. Classical examples are e-learning or training applications, simulation or computer games. New additional application areas are for instance home entertainment systems or infotainment systems in cars.

Such interactive multimedia applications are often developed using authoring tools such as *Adobe Flash*, which includes the programming language *ActionScript*. Such tools provide excellent support for the creative development

tasks. However, they lack of support for structuring the application. The ActionScript code can be scattered all over the application and is very difficult to maintain. Furthermore, there is very low support for teamwork and for coordination between the different developer groups for user interface design, software design and media design. The need for a better support of software engineering principles into multimedia application development is clearly stated by various publications in this area (e.g. [1]).

To address this issues, we propose in [2, 3] a modeling language for model-driven development of multimedia applications called *Multimedia Modeling Language (MML)*. Our idea is to combine this approach with the advantages of the existing tool. Therefore during the design phase only the overall structure and behavior of the application is specified in the MML models. Detailed behavior and concrete visual design should however be created in the authoring tools. We achieve this by generating code skeletons from the MML models which can be directly loaded into the authoring tools for the implementation phase. The code skeletons contain placeholders which have then to be filled out and arranged within the tool. In this paper we demonstrate the feasibility of this concept using as target platform the Flash authoring tool.

The paper is structured as follows: In section 2 we briefly summarize MML. Section 3 introduces the target platform, Adobe Flash, and presents an overview on the main ideas of the approach. Section 4 elaborates a suitable code structure for Flash applications, summarizes the transformation and shows how the resulting documents are processed within the authoring tool.

2 MML

The *Multimedia Modeling Language (MML)* is a platform-independent language for model-driven development of multimedia applications. It supports a design phase for multimedia applications and allows generating code skeletons for different platforms. The language bases on UML 2.0 and integrates concepts from different approaches in user interface and multimedia modeling. Based on the results of requirement analysis - like user task models [4] and storyboards - four kinds of models are provided: *structural model*, *scene model*, *abstract user interface model* (which is enhanced to a *media user interface model*) and *interaction model*. In the following we briefly summarize the MML models referring as example to a Jump'n Run gaming application like those at [5]. For further details on MML please see [2, 3].

The structural model describes the structure of the application logic (domain model) in terms of an extended UML class diagram. The classes from the domain model are referred to as *application entities*. They can be associated with *media components*. For example a character in a Jump'n Run game is represented by an animation. If required, the inner structure of the media components can be defined. This is only necessary when its inner structure is relevant for other parts of the application. An example is a character in a Jump'n Run game: its legs should be animated when the character moves. Thus, the legs have to be realized

as moveable parts of the animation and in some cases it must also be possible to access them from the application logic. As such issues often concern different developer groups – usually the software designer and the media designer – it is important to define them in the model.

The scene model defines the *scenes* of the application and the transitions between them in terms of an adapted UML state chart. A scene represents a specific state of the application’s user interface and is an abstraction of the screen concept in graphical user interfaces. Scenes in a Jump’n Run game are for instance the menu, the game and the highscore. Through their dynamic character caused by temporal media objects, multimedia scenes have an inner state specified by attributes as well as operations to affect their state. In particular, they have so-called *entry-operations* to initialize the scene and *exit-operations* to clean it up.

The abstract user interface model describes for each scene the user interface in terms of *abstract user interface components (AUI components)*. For the AUI components in MML we reuse the concepts provided by user interface modeling approaches (e.g. [6, 7]). The set of AUI components currently supported in MML includes *input component*, *output component*, *action component*, and different specializations of them. The AUI components required within a scene can be derived for instance from task models and are usually specified by the user interface designer.

As a core concept of MML we enhance the abstract user interface with relationships to the media components from the structural model: some of the AUI components could be realized by one or more of the media components. Obviously, often output components are realized by media components. But also input components can be realized by media components as e.g. an animation can be clicked or dragged and dropped. Furthermore, the abstract user interface model is enhanced with *sensors*. A sensor represents an event caused by a temporal media component, like collision sensors for animations (triggering an event when an animation is moved over another object on the screen) or a time sensors for a videos triggering an event when the video has reached a specific point on its timeline. The enhanced abstract user interface model is referred to as *media user interface model*.

Finally, the interaction model describes for each scene how the AUI components and the sensors from the media user interface model trigger operations from the structural model. This model is an adapted UML activity diagram where the actions are restricted to operations calls.

3 General Approach for the Target Platform

For the transformation from MML models to Flash code skeletons we consider the concepts from model-driven development (see e.g. [8]), such as explicit meta-models (to define the models) and explicit, modular transformations between them. MML is defined using a MOF-compliant metamodel. We use the *Eclipse Modeling Framework (EMF)* for its implementation. The transformations are

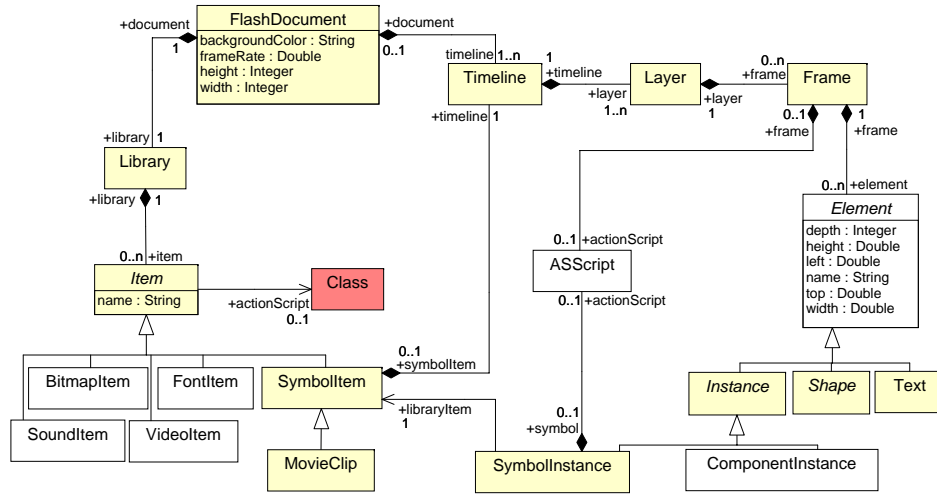


Fig. 1. Extract of Flash metamodel including the main elements of Flash documents.

defined using the *Atlas Transformation Language (ATL)*, a declarative language close to the OMG standard QVT (Queries, Views and Transformations) [9].

The transformation is performed in two steps: first, the MML models are transformed to Flash models. Therefore we specify a Flash metamodel which is presented in this section. The actual mapping from the platform-independent MML concepts into the platform-specific concepts of Flash is performed during this transformation. In a second step, we transform the Flash model into the final code skeletons. This is mainly a straightforward transformation. However, it is more complex than conventional code generation (like transformation from a Java model into Java code), as we aim to generate files for the Flash authoring tool. The resulting files can be directly loaded and processed in the tool using its sophisticated support for the creative design tasks. This requires on the one hand strict compliance to the authoring tool’s internal document structures and on the other hand a solution how to produce the corresponding binary files.

In this section we summarize the capabilities of the Flash authoring tool and the resulting general structure of Flash applications. The main concepts (in the following denoted in *italics*) are reflected in the simplified extract of our Flash metamodel denoted in figure 1. Afterwards we explain our general approach for creating code for the authoring tool. On that base we propose in section 4 a mapping from MML models to a suitable Flash application skeleton.

The Flash authoring tool was originally developed for the creation of graphics and animations. The tool is timeline-based, i.e. the temporal dimension of animations and behavior is represented by a *timeline* consisting of several *frames*.

A frame owns a two-dimensional space (called *stage*, not part of the meta-model) where 2D vector graphics (*shapes* and *text*) and other media objects can be placed. A third dimension (z-axis) is realized by *layers* to define which

object is the topmost when several objects overlap each other on the stage. An animation means that some graphics changes (e.g. its position) over the time, i.e. over the frames in a timeline. A *symbol* is complex graphical object (often the term *movie clip* is used as synonym as movie clip is the most important type of symbol). A symbol contains a timeline which can contain (in its frames) any content as the main timeline. This means that a symbol may contain any complex content, even symbols and animations. Thus, symbols and animations can be hierarchically nested in arbitrary depth. Each Flash document contains a *library* which contains all media objects of the document. When a symbol is created or any media object is imported into the authoring tool, it is automatically added to the library. The *items* in the library can be instantiated multiple times in one or more frames on the timeline. An *instance* usually has an instance name and a location within a frame.

Since version 4 a scripting language is included in Flash, called *ActionScript* which continuously evolved. In Flash MX 2004 ActionScript 2 was introduced which supports the object-oriented concept of classes. Classes have to be specified in separate ActionScript class files. In particular, it is now possible to attach ActionScript classes to symbols in the library of a Flash document. This is a very interesting opportunity, as symbol and associated code then build together a complex object consisting of programming logic and a (possibly very complex) visual representation. The associated ActionScript class has automatically access to all properties of the symbol, as if they were class properties, including visual elements nested inside the symbol. Furthermore events on the symbol (e.g. mouse clicks) can be processed in the class by just specifying corresponding event handler operations. Such a connection between symbols and ActionScript classes is an important concept which we intensively use in our generated code (see section 4).

The file format for the flash documents is a proprietary binary format with the file extension *FLA*. For execution the files are compiled into *SWF* files which run within the Flash player available as plugin for Browsers. SWF is an open format, but as it is a compiled format SWF files can not be edited comfortably within the authoring tool. Hence, for our purposes we aim to generate FLA files.

To solve the problem of creating the proprietary FLA files, we use the mechanism of extensions for the Flash authoring tool. They must be specified in JavaScript and allow to automate every action, which can be done manually in the authoring tool, e.g. creating symbols. For that purpose the tool provides a kind of document object model, similar to that in browsers for HTML documents. These scripts must have the file extension *JSFL* and can be executed either within the authoring tool or from the command line (if the Flash authoring tool is available in the system). We use this mechanism to generate FLA files by generating a JSFL file which can be executed on the command line and then creates the FLA content according to the Flash model (figure 2).

A core problem in Flash is the low support for structuring the applications. The program flow of the application can be determined for instance by ActionScript code, by the timeline or by a combination of both. ActionScript code can

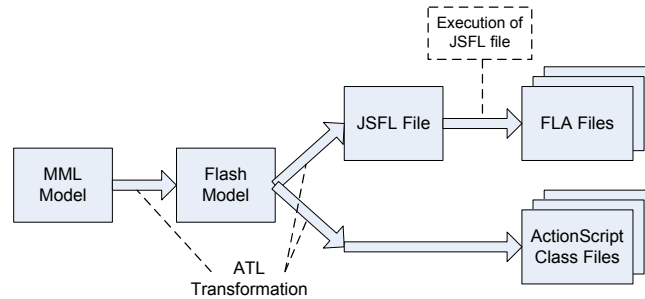


Fig. 2. Approach for the overall transformation

be attached to symbols, symbol instances, and frames. The sources on Flash in the literature and in the web provide various frequently used patterns for many different problems in small scale, but it exists no common solution for the overall structure of Flash applications. An important contribution into this direction is provided by [10] who applies several object-oriented patterns in Flash – e.g. the Model-View-Controller pattern (MVC) – and presents a framework for the overall application structure. However, this approach is restricted to ActionScript code and completely omits the usage of the authoring tool, and is hence not suitable for our purposes. On the other hand, the feedback in the web (e.g. in forums like <http://flashforum.de>) on books like this as well as the latest changes in Flash provided by Adobe show the general demand for a better support of software engineering principles in tools like Flash.

4 Transformation and Resulting Flash Code Skeletons

In this section we describe how to transform the platform-independent MML models into useful code skeletons in Flash. As described in section 3, the literature on Flash provides various different patterns, but there is no common solution for an overall structure of Flash applications which includes both: ActionScript and the features of the authoring tool. Thus, two issues have to be addressed: first we need to identify a suitable structure for Flash applications. Then, the concrete mapping from MML model elements into this structure has to be defined.

Our proposed structure is based on the following considerations: The most important requirement for the Flash application structure is the usage of the authoring tool for creating and editing visual objects. Hence we generate FLA files which contain placeholders (annotated rectangles, see figure 5) for the media components and the AUI components. Besides, the application structure should be well-structured using common concepts, to avoid restriction to specific purpose or specific size and to enable an easy understanding of the generated code. Thus, we use object-oriented ActionScript code for the non-visual parts

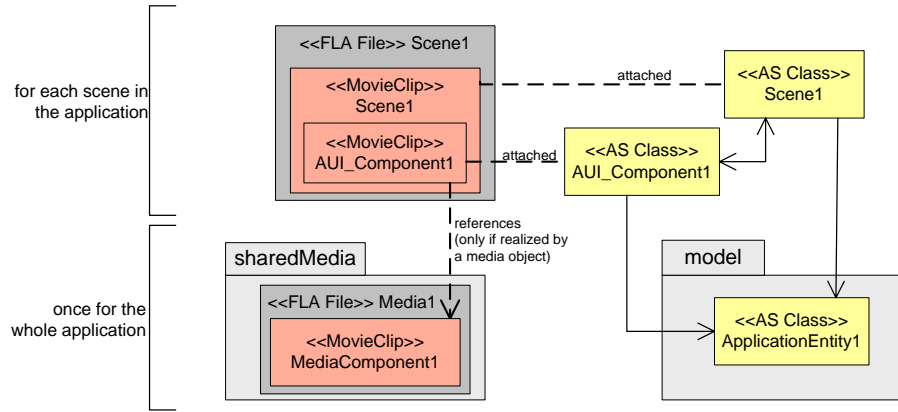


Fig. 3. General structure of Flash applications resulting from the transformation. The names of the artifacts indicate the MML model elements which they result from.

of the application. We make use of the ability of ActionScript 2 and place all ActionScript code into separate class files. As proposed e.g. by [10], we use the MVC-pattern to structure the ActionScript code. For the connections between the visual elements in the authoring tool and the corresponding ActionScript code we apply the ability to attach ActionScript classes to movie clips. User interface objects of others kinds are just encapsulated into movie clips. To support teamwork, we divide the FLA part of the application into many small FLA files. Efficient version management of a single FLA file is usually not possible as FLA is a binary file format. To support development of large applications we provide a package structure for the ActionScript classes and a folder structure for the FLA files. Figure 3 shows an overview on the resulting structure.

The element names in figure 3 indicate the MML model elements where they are derived from during the transformation. The ActionScript classes for the application entities contain the class properties derived from the MML class diagram. For operations we generate only the operation signature, as the operation body is not specified in MML. We believe that the operation bodies are specified more efficiently directly in the target language manually, using the platform-specific constructs and libraries. The classes generated from application entities correspond to the ‘model’ in terms of the MVC-pattern. The ActionScript classes for the scenes contain operations which perform the transitions between the scenes according to the MML scene model. The ActionScript classes for the AUI components contain event handler operations (depending on the type of AUI). They correspond to the ‘controller’ in terms of the MVC-pattern.

For each media component in MML we generate a separate FLA file containing a movie clip in its library which encapsulates a placeholder. The movie clip has attached a name which can be used to refer on it from other files. This ensures that media component can be reused multiple times within an application, as this is possible in MML. If the media component is kind of graphics

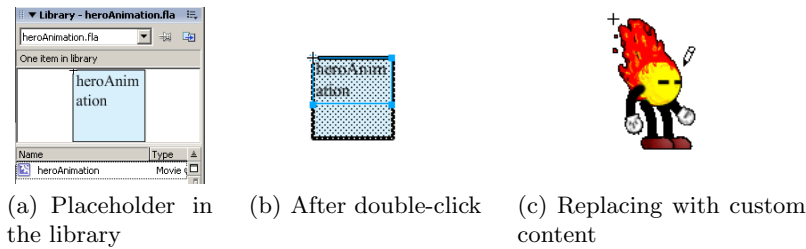


Fig. 4. Replacing the movie clip generated for the media component `heroAnimation`.

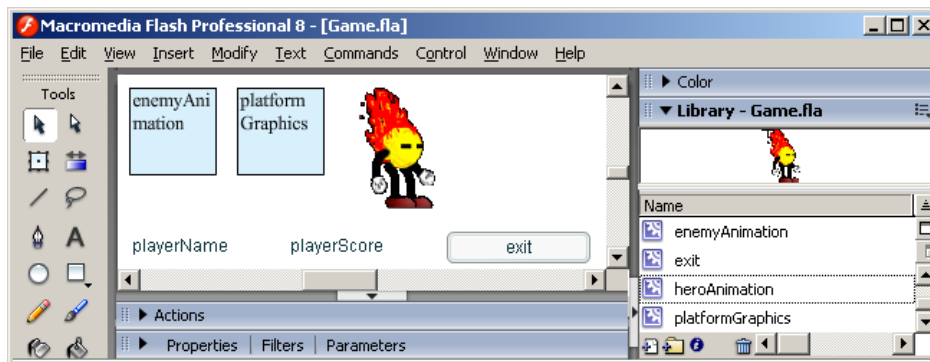


Fig. 5. Screenshot from the Flash authoring tool showing a FLA document generated for the scene `Game`. (The window is reduced to the most important elements.)

or animation the placeholder will usually be filled out directly in the Flash authoring tool. For instance, for an animation `heroAnimation` in a Jump'n Run application a FLA document `heroAnimation` is generated which contains in its library a movie clip `heroAnimation` containing a placeholder (figure 4(a)). The generated movie clip can be edited in the authoring tool as easily as any other manually created movie clip: a double-click on the movie clip opens its content on the stage (figure 4(b)) where it can be replaced by any graphics or animation using the authoring tool's various editing capabilities (figure 4(c)). Other media objects (which can not be created in Flash) will be imported from the file system into the movie clip.

The FLA files generated for the scenes contain the actual user interface of the application (the different 'screens'). They contain the elements generated for the AUI components from the MML model. Figure 5 shows as an example a screenshot of the Flash authoring tool after loading the FLA document generated for a scene `Game` of a Jump'n Run application. Each AUI component is represented by a movie clip (in the library and on the stage) which encapsulates its specific content. This allows us to directly associate it with the corresponding ActionScript class.

AUI components which are not realized by media components are mapped to conventional Flash widget components (located into the encapsulating movie clip). In this case the encapsulating movie clip has no own visual representation on the stage beside the contained widgets. The widgets are labeled with the element name. In figure 5, there are three (invisible) movie clips: one for the output component `playerName` containing a generated text label, one for the output component `playerScore` also containing a generated text label, and one for the action component `exit` containing a generated button. The movie clips representing the AUI component can encapsulate multiple widgets if necessary, for example a text field and a related text label. As explained in section 3 the elements nested into a movie clip can be accessed from the associated ActionScript class as if they were class properties.

If the AUI component is realized by a media component (in the MML model) the generated code uses another ability of the Flash authoring tool: to reuse a movie clip in multiple documents it can be referenced by other movie clips in other FLA documents. In this case the destination movie clip retains its original name and properties, but its contents are replaced with those of the referenced movie clip. Changes in the referenced movie clip appear also in the referencing movie clip. We use this mechanism to reuse the movie clips generated for the media components (e.g. `heroAnimation` in figure 4) in one or more scenes. For instance the scene `Game` in figure 5 contains a movie clip `heroAnimation` which references the `heroAnimation` from `heroAnimation.fla`. In the screenshot, the referenced `heroAnimation` has already been edited while the referenced `enemyAnimation` and `platformGraphics` currently still contain their default placeholder rectangles.

5 Conclusion

In this paper we present a transformation from MML, a language for model-driven development of interactive multimedia applications, to code skeletons for the widespread and professional authoring tool Adobe Flash. Further technical contributions of the paper are the MOF-compliant Flash metamodel and the proposed general structure for Flash applications. As they are independent from the modeling approach, they can be reused for other projects which aim to make use of the Flash authoring tool, e.g. web-engineering approaches which aim to generate rich internet applications (e.g. [11]). Currently, approaches in this area usually use e.g. frameworks like Flex, but they do not support individual user interfaces created in the Flash authoring tool.

Our approach bases on existing concepts from the literature where possible. In particular, we use the abstract user interface model which is common to many approaches in the field of user interface modeling [12]. Thus, it is possible to combine the multimedia-specific aspects from our approach e.g. with concepts for context-sensitive user interfaces as presented e.g. in [6]. Our work also bases on concepts from [13], an existing modeling approach for multimedia applications.

However, to our knowledge none of the approaches aims for generation of code skeletons for an authoring tool like Flash.

MML and the Flash metamodel are implemented using the *Eclipse Modeling Framework (EMF)*. Currently no custom MML editor exists but there is an extension for the UML tool *MagicDraw* which allows creating MML models. The transformations are specified with ATL (see section 3). First user test with the presented concepts were performed in several student projects, mainly in the lecture “multimedia programming” where students developed in teams of 5 to 6 persons (relatively complex) multimedia applications with MML (see [5]). The lessons learned from these practical projects are already integrated into the current version of MML and the Flash code structure.

In all, the paper provides a general proof of concept for the integration of models and authoring tools and shows the required level of abstraction for the models. This results in a combination of the strengths of both technologies: well-structured applications and better coordinated cooperation of developers through models as well as excellent support for the creative design by established authoring tools. In general, the idea of integrating modeling with more informal techniques and tools for the creative development tasks might be another step towards a better integration of software engineering and human-computer interaction.

References

1. Hirakawa, M.: Do Software Engineers Like Multimedia? In: IEEE International Conference on Multimedia Computing and Systems (ICMCS). IEEE (1999)
2. Pleuß, A.: Modeling the User Interface of Multimedia Applications. In: MoDELS 2005. Volume 3731 of LNCS, Springer (2005)
3. Pleuß, A.: MML: A Modeling Language for Interactive Multimedia Applications. In: 7th IEEE International Symposium on Multimedia (ISM 2005). IEEE (2005)
4. Paternó, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: Interact’97. Chapman & Hall (1997)
5. University of Munich: Lecture Multimedia-Programmierung, Summer Term 2006, <http://www.medien.ifl.lmu.de/studiengang-neu/galerie/mmp-ss06/> (2006)
6. Van den Bergh, J., Coninx, K.: Towards Modeling Context-Sensitive Interactive Applications. In: SoftVis 2005. ACM Press (2005)
7. Constantine, L.L.: Canonical abstract prototypes for abstract visual and interaction. In: DSV-IS. Volume 2844 of LNCS, Springer (2003)
8. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley (2003)
9. Jouault, F., Kurtev, I.: On the architectural alignment of atl and qvt. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), ACM (2006)
10. Moock, C.: Essential ActionScript 2.0. O’Reilly Media (2004)
11. Bozzon, A., Comai, S., Fraternali, P., Carughi, G.T.: Capturing RIA concepts in a web modeling language. In: WWW 2006, ACM (2006)
12. 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: TAMODIA, INFOREC Publishing House Bucharest (2002)
13. Sauer, S., Engels, G.: Uml-based behavior specification of interactive multimedia applications. In: HCC’01, IEEE (2001)