TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik
Institut für Systemarchitektur, Professur für Rechnernetze
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill.

DIPLOMARBEIT

Modellgetriebene Entwicklung von Web2.0-Applikationen aus AUI-Beschreibungen

Vorgelegt von: Andrej Kabachnik
geboren am: 21. September 1984 in: Moskau

zum
Erlangen des akademischen Grades

DIPLOMINGENIEUR
(Dipl.-Ing.)

Betreuer: Dipl.-Inf. Marius Feldmann

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage dem Prüfungsauschuss der Fakultät Informationssystemtechnik eingereichte Diplomarbeit zum Thema

Modellgetriebene Entwicklung von Web2.0-Applikationen aus AUI-Beschreibungen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 30.09.2009

Unterschrift

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Anfertigung dieser Diplomarbeit unterstützt haben.

Ein ganz besonderer Dank gilt meinem Betreuer, Herrn Marius Feldmann, der mich während der gesamten Arbeit durch seine Anregungen und seine Geduld unterstützt hat, und mir insbesondere bei der schriftlichen Ausarbeitung sehr nützliche Tips gegeben hat.

Des Weiteren möchte ich mich bei Frau Natalia Arefyeva für ihre unermüdliche moralische Unterstützung bedanken.

Nicht zuletzt geht ein herzliches Dankeschön an meinen Vater, Prof. Dr. Nikolay Kabachnik, der mir bei der Korrektur der schriftlichen Arbeit geholfen hat, und an meine beiden Eltern, die dieses Studium überhaupt ermöglicht haben.

Zusammenfassung

In den letzten Jahren sind plattformunabhängige Abstrakte User Interfaces (AUIs) zu einem wohl bekannten Werkzeug zur modellgetriebenen Erzeugung von UIs geworden. Dieser Ansatz ist besonders vorteilhaft für Web-Applikationen, wo die Benutzeroberfläche von der zugrundeliegenden Businesslogik physikalisch getrennt ist. Mit der wachsenden Verbreitung von hoch-interaktiven Web2.0-Applikationen, weicht allerdings das alte Seiten-Paradigma einem neuen Desktop-ähnlichen Interaktionsmodell, welches von Technologien wie AJAX ermöglicht wird. Jedoch gibt es derzeit noch keine weitverbreitete Methodologie zur Codegenerierung für AJAX-UIs aus plattformunabhängigen Modellen. Existierende Ansätze sind entweder zu aufwändig und sehen mehrere Zwischenmodelle vor, oder benutzen spezialisierte Modelle. In dieser Diplomarbeit wird eine neue Methode zur Generierung von AJAX-UIs basierend auf einer modernen abstrakten UI-Beschreibungssprache MARIA XML vorgestellt. Mit Hilfe der entwickelten Konzepte wird eine prototypische konfigurierbare Codegeneratorumgebung implementiert, die in der Lage ist Programmcode für verschiede AJAX-Toolkits aus einem einzigen AUI-Modell zu produzieren. Schließlich werden die Ergebnisse evaluiert und auf Funktionalität, Universalität, Benutzerfreundlichkeit und Performance untersucht.

Abstract

In the past couple of years, platform independent abstract user interfaces (AUIs) have become a well-known tool for model-driven generation of application UIs. This approach is especially beneficial for web applications, where the UI is physically separated from the underlying business logic. However, with the growing popularity of highly interactive Web2.0-applications, the old page-based paradigm gives way to a new desktop-like rich interaction model provided by technologies like AJAX. Nevertheless, there is still no wide spread methodology of AJAX UI generation based on platform independent models. Existing approaches are either too complicated, involving multiple intermediate models, or use specialized models. In this thesis a novice approach to AJAX UI code generation based on a modern abstract UI description language MARIA XML is proposed. Elaborated concepts are used to implement a sample configurable code generation environment, capable of producing code for different AJAX toolkits from a single AUI model. Finally, the results are evaluated with respect to usability, versatility, functionality, and performance.

Inhaltsverzeichnis

1 Introduction **1**
- 1.1 Main goals . . . 2
- 1.2 Requirements . . . 3
- 1.3 Outline . . . 4

2 Terminology **5**
- 2.1 Model-Driven Architecture . . . 5
- 2.2 Metamodels and Models . . . 7
- 2.3 MDA toolkits . . . 7
- 2.4 Abstract user interface descriptions . . . 8
- 2.5 Rich Internet applications and interfaces . . . 10
- 2.6 AJAX . . . 10
- 2.7 AJAX frameworks and toolkits . . . 11

3 State of the art and problem statement **14**
- 3.1 Related work . . . 14
- 3.2 Model-driven UI development for web 2.0 . . . 19
- 3.3 AUI and CUI . . . 20
- 3.4 Code generation from AUI . . . 21
- 3.5 MARIA XML overview . . . 23
- 3.6 Problem statement summary . . . 25

4 The Concept **27**
- 4.1 Model-driven UI generation . . . 27
- 4.2 MARIA XML AUIs . . . 28
  - 4.2.1 Presentations . . . 29
  - 4.2.2 Connections . . . 30
  - 4.2.3 Interactors . . . 32
  - 4.2.4 Data model . . . 35
  - 4.2.5 Event model . . . 36
  - 4.2.6 Generic script . . . 39
  - 4.2.7 External functions . . . 41
  - 4.2.8 Continuous updates . . . 42

4.3 Extensions to MARIA XML . . . 43
4.3.1 Styling information in the AUI . . . 43
4.3.2 Adding the content . . . 44
4.3.3 Semantic interconnections between UI elements . . . 45
4.3.4 Drag and drop . . . 46
4.4 The code generator . . . 47
4.4.1 Code generation templates . . . 48
4.4.2 Customizing the generator . . . 49
4.4.3 All-in-one templates . . . 50
4.5 Concept summary . . . 51

5 Implementation **53**
5.1 Use case: a simple cookbook . . . 53
5.2 openArchitectureWare . . . 54
5.3 MARIA XML as an ECore model . . . 56
5.4 MARIA XML data model . . . 56
5.5 Code generation templates . . . 56
5.5.1 Template structure . . . 57
5.5.2 Template variants . . . 59
5.6 JavaScript . . . 60
5.6.1 Toolkit independent utilities . . . 61
5.6.2 JavaScript toolkit adapters . . . 63
5.7 Sample adapters . . . 64
5.7.1 Semantic content and styling . . . 65
5.7.2 Multimedia . . . 65
5.7.3 Semantic connections . . . 66
5.7.4 On-the-fly input validation . . . 66
5.7.5 Events . . . 67
5.8 Sample application . . . 68
5.8.1 Server part . . . 68
5.8.2 UI structure . . . 69
5.8.3 Nested presentations and partial UI updates . . . 71
5.8.4 Validating the registration form . . . 73
5.8.5 Application variants . . . 73
5.9 Implementation summary . . . 74

6 Evaluation **75**
6.1 Primary results overview . . . 75
6.2 Configurable code generator vs. CUI . . . 76
6.3 Efficiency and performance of generated UIs . . . 78
6.4 Main limitations . . . 81

6.5 Outlook 82

7 Conclusions **84**

A Appendix **86**
A.1 Content of enclosed CD 86
A.2 Setting up the code generator 86
A.3 Testing the generated UI 87

Bibliography **88**

1 Introduction

During the past two decades the Internet evolved to a whole new world which has come to play an important role in our everyday life. The number and diversity of web applications exploded. Following the growing demands towards efficiency, complexity, and flexibility of Internet based applications, new approaches for web software development have emerged.

Recently, a broad collection of new trends in the development of web-based application user interfaces have appeared under the Web 2.0 umbrella, changing the classical web interaction model significantly. One of the most prominent technologies associated with Web 2.0 applications is AJAX (Asynchronous JavaScript and XML) [Gar05], in which a combination of JavaScript and Document Object Model (DOM) manipulation, along with asynchronous communication is used, to achieve a high level of interactivity on the Web. What enables AJAX UIs to be more responsive and interactive compared to the traditional page-based HTML is the ability to exchange messages with a server behind the scenes and to update individual components of the user interface as required. The intriguing feature of AJAX and what makes it different from classic Web interactions is that this communication occurs without the need for the entire web page to reload. Through such a possibility, giving users the richness and responsiveness of desktop applications has become feasible, and accordingly, many major web applications have employed AJAX in parts or all of their user interfaces. It has even become possible to port popular desktop applications, such as electronic maps (like Google maps[1]) or document processing software (like Google docs[2]) to the web.

With AJAX popularity exploding, a vast amount of widget libraries, toolkits, and frameworks of all kinds emerged. They enable software developers to quickly build impressive complex UIs for their web applications by using ready-built functions and chunks of code. The rapid development of new UI elements, styles, and standards and the growing complexity of the back-end business logic have led to increased separation of these two domains. Indeed, the look-and-feel of a web application can be changed by using other UI controls, widgets, and styles while the back-end, that runs the application logic on server-side, remains untouched.

In many cases, different UI variants for the same application need to be created for different browsers, different target devices, etc. Additionally, the problem of choosing the

[1] Google maps, http://maps.google.com
[2] http://docs.google.com

right toolkit or widget library is not to be underestimated with so many choices available. One way to tackle these challenges is by abstracting from the implementations through a Model-Driven Engineering [Sch06] approach, in which an application is defined in a modeling language and the corresponding code is (semi-)automatically generated. This approach enables developers to define the application (or it's UI) once and easily port it to different frameworks or devices, or to update an implementation to use a new technology. Modern modeling tools can generate the entire code from models which saves a lot of time and eases the development significantly. Since new technologies and frameworks are emerging in an overwhelming pace, this approach becomes vital for keeping large applications up to date.

Following the general trend in software development set by the UML modeling language, special modeling languages for web applications were introduced. While modeling languages for traditional page-based web applications like WebML are already well developed and widely used, they still lack the possibilities for detailed UI descriptions to model an AJAX user interface. To solve this problem, new dedicated abstract UI description languages started to emerge. This trend is greatly supported by the growing popularity of web-service-based applications, where the separation of UI and business logic is maximized.

Although model driven UI development gives the possibility to generate UI code from the model automatically, some manual refinement is still needed to make the UI user friendly. The traditional approach to model-driven UI development defines an intermediate step between the abstract UI (AUI) and the final UI (FUI) called concrete UI (CUI). It is the CUI, where the spatial arrangement of elements, their styles, sizes, and colors are defined. This generally requires a lot of human work even though special assisting tools can be provided. Nevertheless, the model-driven approach saves considerable amounts of time and effort.

In this work an attempt to go one step further towards fully-automated UI generation for AJAX applications shall be made. Based on the abstract UI description language MARIA XML [PSS09] the concept of direct AJAX code generation from AUI models will be developed and evaluated. A sample implementation of a code generator environment capable of producing code for different AJAX frameworks from the same model will be presented to verify the elaborated concepts.

1.1 Main goals

A lot of research had been done on model-driven UI development for web applications using the traditional AUI-CUI-FUI approach. Is the creation of the CUI really a mandatory step, though? Its main advantage - fine-granulated device and platform adaptation - may

not even be needed in the context of the web. On the other hand, the code generator itself can take care of this to some extent. And in the specific case of web applications, the layout and look&feel is mostly defined separately by cascading style sheets (CSS).

On the other hand, since the CUI mostly requires human attention, skipping it may speed up UI generation. The model-to-model transformation rules used to generate the CUI itself would not need to be created either, and there would be no need to worry about consistency when changing the AUI. If it would be possible to generate code from the AUI directly, almost the entire development process could be automated.

Throughout this work the following main objectives are pursued.

- Integration of fully automated AUI-to-FUI transformation into the development process of AJAX-based Web 2.0 applications.
- Evaluation and elaboration of the abstract UI description language MARIA XML with respect to AJAX-based applications.
- Definition of borders of the AUI-based approach to user interface development for Web 2.0 applications.
- Comparison between the estimated efforts for AUI-based development and manual coding.

1.2 Requirements

Since the term AJAX stands for a very broad and not really clearly defined collection of technologies, we shall extract some of the most important properties of AJAX-applications here. For the scope of this work, these properties will define an AJAX-application as such.

- XHTML + JavaScript as client-side technology.
- Immediate response to user actions (where desired).
- Partial content refreshes.
- Asynchronous communication (streaming).
- Customizable appearance (CSS).
- Rich interactions (e.g. drag&drop).

The ultimate goal shall be the generation of an application with the above properties from a MARIA XML abstract UI description with extensions if needed. These extensions, however, shall not effect platform independency if possible.

1.3 Outline

After an intorduction to the used terminology in the following chapter, a state of the art analysis and a brief introduction to MARIA XML will be presented in chapter 3. In chapter 4 a detailed evaluation of the possibilities of code generation from the major elements of the AUI will be followed by a description of the proposed extensions to MARIA XML. After this code generation templates and feature management techniques will be discussed. In chapter 5 a detailed overview of the sample implementation of an advanced code generation environment with modular support for multiple widget libraries will be given. Chapter 6 will evaluate the results, point out advantages and shortcomings of the tools used, and propose possible subjects of future work. Last but no least, chapter 7 will conclude with a compact summary of the entire work.

2 Terminology

2.1 Model-Driven Architecture

Model-driven architecture (MDA) is a software design approach launched by the Object Management Group (OMG)[1] in 2001. It provides a set of guidelines for the structuring of specifications, which are expressed as models. MDA separates business and application logic from underlying platform technology.

As defined by the OMG, MDA provides an approach for, and enables tools to be provided for [MM03, VG08]:

1. Specifying a system independently of the platform that supports it;
2. Specifying platforms;
3. Choosing a particular platform for the system; and
4. Transforming the system specification into one for a particular platform.

In other words, MDA proposes to start a software project by first defining platform-independent models. Subsequently, transformations should be defined to transform these models, possibly in a number of iterations, to other models which are platform-specific. The ultimate step is to generate code from the lowest-level platform-specific models.

MDA formalizes a set of terms to denote the abstraction levels of the models. Important terms are:

PIM a Platform Independent Model (PIM) is a view of a system, exhibiting a specified degree of platform independence so as to be transformable to different platforms of similar type.

PSM a Platform Specific Model (PSM) is a platform specific view of the system. Specific information about the platform may be used in the views for better understanding and implementation of the system.

MDA encourages declaring a PIM using a platform independent language such as UML. By using this approach, developers can ignore secondary details while focusing on the core functionality of the system. When the PIM is modeled sufficiently, transformation

[1] http://www.omg.org/

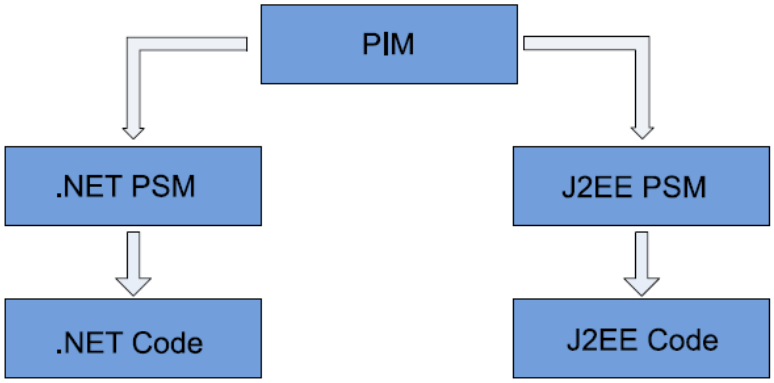

Figure 2.1: Example of PSMs and code being generated for different platforms from a single PIM [VG08]

techniques are employed to transform it to a PSM. Subsequently, a PSM may be used for (semi-)automatic code generation. However, generating code from a PIM directly is also a possibility which shall be discussed later in greater detail. Figure 2.1 demonstrates an example of how a PIM can be used to generate code for two different platforms.

The following is a summary of the main advantages of using an MDA approach for web software development [MM03, VG08].

Comprehension. Prior and during the development of a web system, information on it needs to be communicated and documented. Using an MDA approach means up to date and accurate models of the software system are available.

Evolution. New technologies constantly come and disappear. With MDA the current concept can be shipped to that new 'hot technology' with less effort.

Migration. Possibility to extract models from legacy web applications and generate AJAX web applications from them. Possibly reuse the extracted models for the next new technology.

Maintenance. Modeling software systems prior to their development leads to software which is easier to develop, maintain, and integrate. Having machine-readable designs at ones disposal greatly reduces maintenance costs.

Automation. Through MDA, many parts of the development processes are automated. Best practices can be put to use in an automated fashion. Developers need not be concerned with these and other implementation details and can therefore concentrate on the semantics of the application. The models can go through automatic validation, testing and be checked for inconsistencies.

Integration. Easier integration between different web components when models are

available. For instance, the possibility for an application based on the combination of JSFSpring- Hibernate, JSF-Hibernate or JSF-Spring-EJB to be created in an automated way.

2.2 Metamodels and Models

Metamodels define the language used to build models. If a model is an abstraction of a phenomena in the real world; a metamodel is yet another abstraction, highlighting properties of the model itself.

The Meta-Object Facility (MOF) is an Object Management Group (OMG) standard for model-driven engineering [Wik09]. It is designed as a four-layered architecture. It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by MOF to build metamodels, called M2-models. The most prominent example of a Layer 2 MOF model is the UML metamodel, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects.

Beyond the M3-model, MOF describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the MOF M3-model and UML structure models, MOF metamodels are usually modeled as UML class diagrams. A supporting standard of MOF is XMI [OMG07], which defines an XML-based exchange format for models on the M3-, M2-, or M1-Layer.

2.3 MDA toolkits

MDA toolkits provide a framework, within which metamodels and transformations can be defined to be used in model-driven software development. Two major open-source MDA frameworks are AndroMDA[2] and openArchitectureWare (oAW)[3].

The key difference between them is that AndroMDA is a generator, whereas oAW is an Eclipse-based generator framework. This means that AndroMDA comes with a bundle of ready-to-use metamodels and templates, making it easy to get started with simple projects as long the provided templates are sufficient. oAW, on the other hand, focuses on supporting the user in creating his own metamodels and generators. In contrast to AndroMDA, it does not require the use of UML, but allows one to use arbitrary model

[2] http://www.andromda.org/
[3] http://www.openarchitectureware.org/

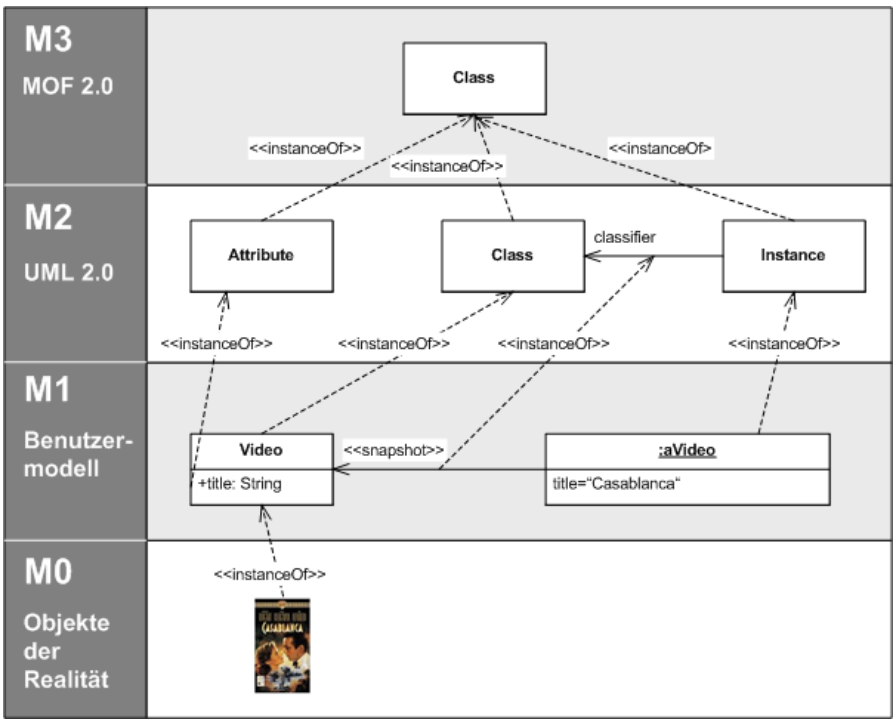

Figure 2.2: Illustration of the Meta-Object Facility [Wik09]

formats with special support for UML, EMF, XML, and some others. A number of prebuilt oAW workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models, and then finally, for generating code.

2.4 Abstract user interface descriptions

Abstract user interfaces (AUIs) are platform independent models of user interfaces (UIs). Just like any other PIM, AUIs can be transformed to a model for a concrete target platform or programming language - a CUI (concrete user interface) which, in turn, can be used to generate code - the final UI (FUI). However, while most modeling techniques are presently used to describe application logic, data structures, etc., AUIs focus exclusively on the user interface. In fact, the underlying business logic is separated completely and even can be fully replaced without any effect upon the UI.

AUIs are described in abstract user interface description languages (which make up the corresponding metamodels). These generally are declarative mainly XML-based

markup languages. They are intended to be used by designers rather than developers, thus allowing them to design a usable UI without the need of high programming skills. Graphical tools can be developed to assist in the design process. In this case, the designer would not even need to have extensive knowledge of the description language itself.

While the separation of the UI and the application logic is clearly an advantage in many senses, it implies the use of a well-structured interface to the application's back-end. Should this interface remain unchanged, the two parts of the application, which it connects, can indeed be altered independently.

In most cases the AUI is used to store very generic information about UIs: general element types, interaction controls, etc. It is the CUI where spatial arrangement is introduced and element's display variants, interconnections, and behaviors are specified precisely. This is where the abstract UI is adapted to the concrete demands of the target device and platform. Powerful platforms like PCs may need more complex interfaces with richer functionality than mobile devices with limited screen resolutions and computing power. While it makes sense to show high-resolution images, colorful animations and multimedia content on a stationary PC, it may rather be a good idea to downsample the images and to use thumbnails for multimedia content on a mobile phone to save bandwidth.

Large parts of the CUI can be obtained from the AUI by defining a set of rules for model transformation. In the majority of cases, however, some serious manual editing is needed to create a good CUI. The FUI can then be generated full automatically, using a code generation template for a specific programming language. The AUI itself can often be derived from the model of the back-end application. So each step in the UI generation process can be at least partially automated.

The use of automated transformations and code generation eases the development significantly. For example, new UI versions for different devices or platforms can easily be added by adding another CUI and generating code from it. And even if the back-end interface should change, the changes will quickly propagate to the final interfaces once the AUIs and CUIs have been adjusted.

In the particular case of web applications the model-based approach seems to be especially suitable, because all the business logic is generally handled on server-side, while the client merely shows the corresponding UI. And with the development of new types of web interfaces exploding, it surely is a good idea to automate the adaption of existing web applications to the new technologies, toolkits, and frameworks. Another advantage comes with the increasing popularity of web-service-based applications. Each web service has an abstract interface description (mainly in WSDL) in any case. These descriptions can be used to generate AUIs. It could even become possible to compose an application from existing web services by composing AUIs, generated from their WSDL-descriptions,

and using it to generate the actual code for the UI, thus saving huge amounts of time and development efforts.

2.5 Rich Internet applications and interfaces

"Rich Internet applications" (RIA) is an umbrella term for web applications with desktop-like user interfaces. In contrast to page-based user interfaces, as seen in standard web pages or WAP, rich interfaces provide more interactivity and a fluid control flow. Although the term is quite blurred, it is common to consider the following features some of the keystones of rich user interfaces (RUIs) [MAMRVO06].

- Dynamic data retrieval (streams of data from client to server and vice versa at runtime).
- Perceptive continuity (reduction of page refreshments and freeze conditions).
- Adaptability that is the faculty of responding in autonomous way to the user necessities.
- Strong support for multimedia (capability of managing embedded graphics, video, audio, and streaming).
- A User Interface Language that describes the UI for multiple contexts.
- Push technology (acquiring data that has not been requested by the user in order to update the current information [FZ98]).
- Support for rich interactions like dragging objects, etc.

The fundamental technological difference between a RIA and a legacy web application though is, that the former takes advantage of the clients computing power to render the UI while the latter merely uses the client to present a static page rendered by the server.

2.6 AJAX

AJAX (Asynchronous JavaScript and XML) [Gar05] is a technique used for creating highly interactive Web user interfaces. Using AJAX can be a way to create RIAs (although it is arguable, if an AJAX application is automatically a RIA). The interactivity aspect of an AJAX web application manifests itself in that an AJAX web application can consist of a single page, which does not reload as a result of round trips to the server. A famous example of AJAX can be found in Google Maps[4], where users can seamlessly explore maps as if all the data has already been loaded in the client, while in reality, data is constantly being requested from the servers to satisfy the users' dragging and zooming.

[4] http://maps.google.com

This form of communication is fundamentally different from classic web applications, where any communication with the server involves the whole page being fetched and rendered again in the browser. The possibility to communicate with the server asynchronously provides a richness and responsiveness similar to desktop applications. To achieve this higher level of interactivity, asynchronous communications with the server are combined with dynamic alterations of the user interface. In this manner, a user interface can communicate with the server and only update relevant parts (components) of the page, leading to a fine-grain interaction. On the whole, AJAX incorporates the following techniques:

- The XMLHttpRequest [W3C06] object for asynchronous communication with the server.
- Dynamic alteration of the user interface using DOM [W3C05a]
- Presentation using XML, XSLT, and CSS.
- JavaScript for integrating the previous techniques.

Today many web sites employ AJAX in their user interfaces, though often to different degrees. Below three main categories, as to the extent AJAX is used in a user interface, are presented [VG08]:

1. No AJAX at all. All communication with the server is synchronous.
2. AJAX in parts of the web application. For instance, the photo sharing web application Flickr[5] employs AJAX only for a specific number of features, such as editing the title and label of a photograph.
3. Fully single-page AJAX. For example, the new version of Yahoo! Mail[6] which is an e-mail application with the look and feel of a desktop e-mail agent, such as Mozilla Thunderbird or Microsoft Outlook.

2.7 AJAX frameworks and toolkits

AJAX frameworks and toolkits are collections of JavaScript components designed to ease rapid development of cross-browser JavaScript/AJAX based applications and web sites.

Since AJAX actually consists of multiple different technologies, which are strongly platform dependent, compatibility has always been a great concern. Not only the lack of proper JavaScript support in some (especially mobile) Internet browsers, but also different implementations of the XMLHttpRequest and the DOM cause frequent compatibility

[5] http://www.flickr.com/
[6] http://mail.yahoo.com

issues. Browser-independent AJAX toolkits enable developers to focus on the actual functionality by taking care of most of the compatibility problems and providing additional tools to simplify development of complex applications. They conceal the complexity of implementing Ajax from the developer by providing a simplified programming model.

The Dojo Toolkit is currently one of the most well developed JavaScript toolkits with extensive AJAX support. It offers event control, packaging, CSS-based querying, animations, JSON utilities, etc., and still does not require any additional components to be installed on client-side. Normal JavaScript support suffices. The compact Dojo core can be extended by Dijit and DojoX, which are separate projects offering a wide selection of widgets: from tabs and accordions to complex 2D and 3D drawing, data visualization, and more.

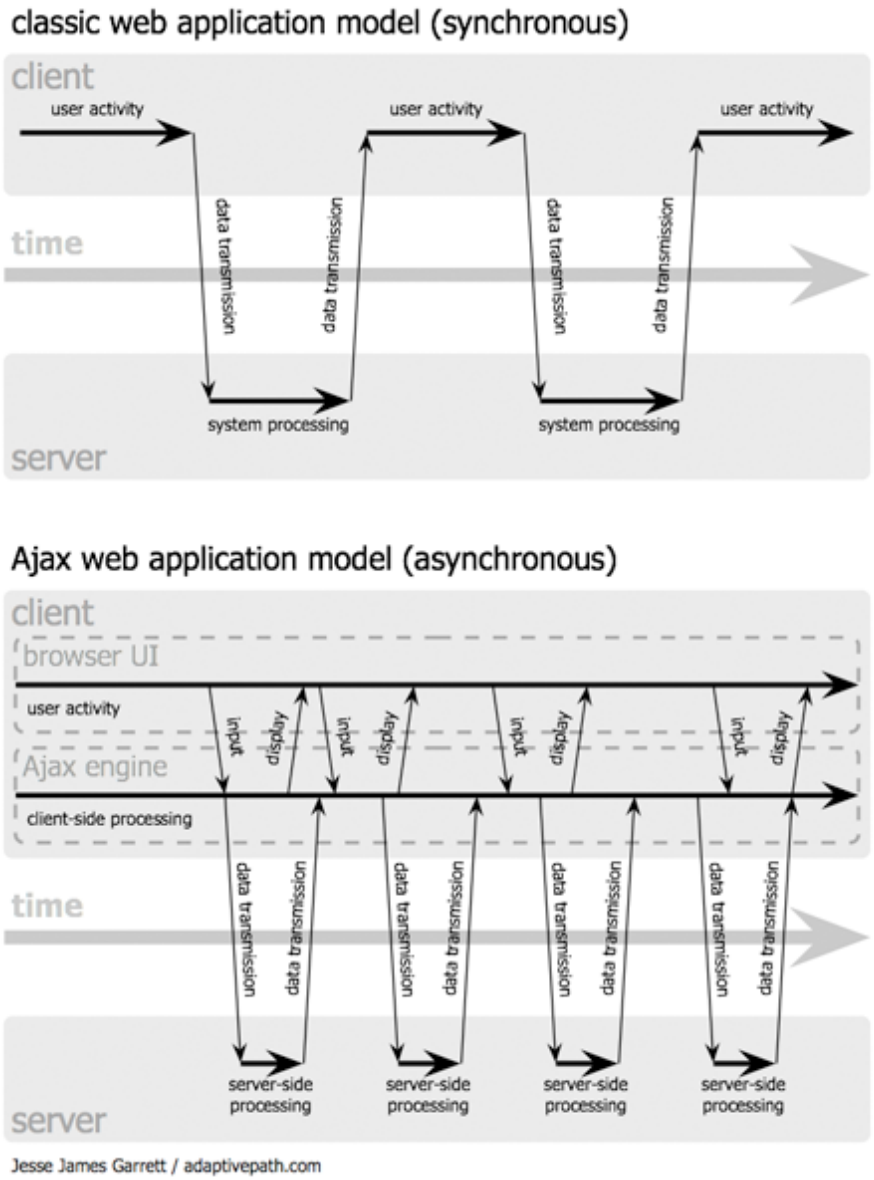

Figure 2.3: Asynchronous communication in AJAX [Gar05]

3 State of the art and problem statement

3.1 Related work

Over the past several years a lot of research and development had been done in the area of MDA for the web. Most of the projects, however, concentrate on describing the entire application: from the underlying data structure over business logic to the final user interface. The majority of them focus mainly on data structures and application logic. These modeling languages are mostly UML-based, following existing standards in model driven application development. On the other hand, as the need for flexible user interfaces, which can be easily adapted to different devices and platforms, grows ever greater, the benefits of abstract UI descriptions draw attention of developers. The model-driven approach to web-based rich user interfaces, however, still remains mostly untouched, since at the time being legacy page-based web applications prevail.

For example, the Web Modeling Language (WebML) [CFB00] is a popular notation for specifying the structure and navigation of legacy web applications. The notation greatly resembles UML class and Entity-Relation diagrams. Presentation in WebML is mainly focused on look and feel and lacks the degree of notation, needed for AJAX web user interfaces. Although the original WebML has been recently extended [BCFC06, TC07] to achieve more RIA design objectives, its presentational possibilities still are not enough to define a full-featured AJAX-UI [PLCSF07]. Conallen [Con03] proposes a UML profile for modeling web applications. This approach is widely referenced as web modeling scheme. However, it also mainly supports the classic multi-page web applications. Koch and Kraus [KK02, KK03, KK06] propose UWE, a UML profile and notations for modeling the navigation and conceptual aspects of a web application. UWE has relatively well developed CASE-tool support, but rich interfaces are presently not supported either, although it is said to be planned for the near future.

A modeling framework aimed at AJAX in particular was introduced by S.V. Vahid Gharavi [VG08]. His work resulted in a plug-in (cartridge) for the code generator AndroMDA. A special UML-based AJAX metamodel was developed, which indeed allows to create models, which can be used to generate code for AJAX applications. However, similarly to the preceding examples, this approach targets the generation of complete applications based on UML models, rather than the UI only. It does not specially address

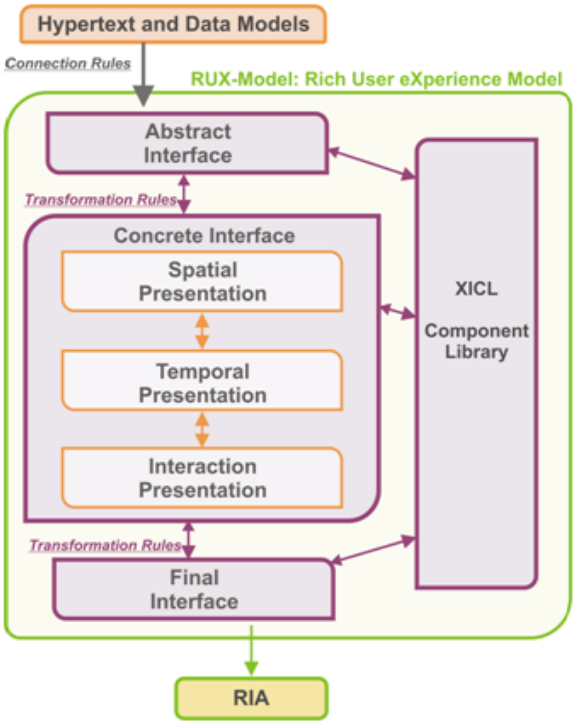

Figure 3.1: RUX-model (http://ruxproject.org)

the look&feel of the UI or advanced widget properties. Apart from this, the use of a special AJAX metamodel makes it less versatile since it is bound to a specific technology.

To cope with further RIA capabilities still uncovered by WebML's RIA extension, another metamodel called RUX (Rich User eXperience) has been proposed in [LPSF07]. In a sense, RUX is a UI description model based on the AUI-CUI-FUI approach, which is meant to be placed on top of an application model. It was developed to be used in connection with WebML, but in theory it could also describe an arbitrary UI. The general structure of the approach is shown in figure 3.1. The abstract interface, imported from the WebML model, can be manually refined to form the device-specific concrete interface. There is a visual tool called RUX-tool[1] to support this transformation. The concrete UI defines spatial arrangement of elements, their temporal behavior and user interaction events. The final UI code can be obtained by code generation based on the concrete UI model.

The combination of WebML and RUX allows the developer to address advanced features of RIAs like distributing data handling between client and server, optimizing communication,

[1] http://ruxproject.org/ruxtool.html

etc. (for details refer to [PLCSF07]). However, this is an advantage of the two models used together, not a property of the RUX-model itself.

UsiXML[2] is a still evolving set of markup languages intended for non-programmers, such as analysts, designers, human factors experts, etc. to easily take part in UI development without the need to acquire special knowledge about the underlying platform. Multiple graphical tools are available for different tasks (development, composition etc.). Possible methodologies to generate RIA user interfaces from UsiXML descriptions have been studied in [MR07]. The author used a bottom-up approach to describe a complex RUI example by means of UsiXML. The elaborated methodology suggests the use of an extended version of ConcurTaskTrees (CTT) [Pat99] as starting point for an iterative creation of an AUI, CUIs and FUIs. The primer focus of the work, however, lies in the automated generation of an AUI from CTTs, which involves component grouping, defining navigation, etc. Model-to-model transformations and code generation were done by XSLT. The result was a XAML-based UI description, but theoretically the same approach could be used to generate an AJAX-application[3].

XAML[4] is another UI description language, which is widely used by Microsoft. In connection with Microsoft's new programmable browser plug-in Silverlight, XAML can be used to describe AJAX-based rich Internet applications. However, installation of the Silverlight plug-in on the client's browser is still required to run the application. The Mozilla Foundation developed its own language: XUL[5]. Originally, it depends on client-side presence of the Gecko engine, however there is also a browser independent alternative: the ZK Direct RIA[6] is an AJAX-based web framework utilizing XUL. ZK actually uses a combination of strongly extended XUL and XHTML to form a new markup language, called ZUL. An Eclipse-based visual editor is provided as part of the ZK Studio - a well developed set of interconnected development tools. ZK's AJAX engine consists of both client (JavaScript) and server (Java servlet) components that communicate with one another using asynchronous connections (figure 3.2). While offering an amazing variety of components and a well built environment, ZK's UI description language lacks flexibility as it has to be used with the ZK framework and cannot be utilized to describe an arbitrary web application UIs. All three languages have good tool support, but the UI needs to be developed manually. There is no wide-spread methodology of generating it based on other models.

IBM's alphaWorks - a web community for early adopters to preview and collaborate on prototype technology - developed a vendor independent XML-based abstract UI

[2] http://www.usixml.org/
[3] Unfortunately the code generation templates are not available for download at the moment
[4] http://msdn.microsoft.com/en-us/library/ms752059.aspx
[5] https://developer.mozilla.org/en/XUL
[6] http://www.zkoss.org/

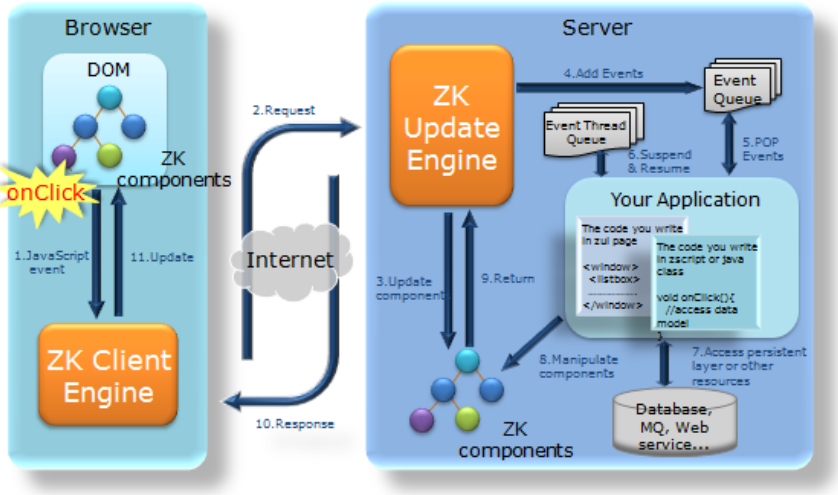

Figure 3.2: ZK framework architecture [ZK]

description language, called AUIML[7] (Abstract User Interface Markup Language). The AUIML Toolkit consisted of an Eclipse-based Visual Builder and both, a Java Swing renderer and a Web renderer (theoretically other renderers could be added). The Visual Builder was used to manually create the panels (pages), which could then be tested through the preview function, provided by the two runtime renderers. The Web renderer was said to be able to produce AJAX-based applications as well. However, this project has been discontinued.

Although the original project was officially dropped, a new publication [LLPS08], which appeared towards the end of 2008, introduced the AUIML Web Toolkit (AWT). The original AUIML Toolkit was redesigned to utilize JavaScript instead of Java, thus producing an application capable of running on an ordinary web browser. An overview of its architecture is presented in figure3.3. The authors used widgets from the Dojo Toolkit for a sample application. However, the status of the project is somewhat unclear, as the original AUIML Toolkit, which it is based on, is not even available for download anymore.

Table 3.1 shows a brief comparison of the key usability properties of AJAX-capable technologies introduced above. The combination of WebML and the RUX-model as well as the ZK-Framework offer the most extensive support for AJAX features. However, while RUX strongly depends on the underlying WebML, ZK's ZUL cannot really be considered an abstract UI description language, but rather a specialized markup language

[7] http://www.alphaworks.ibm.com/tech/auiml

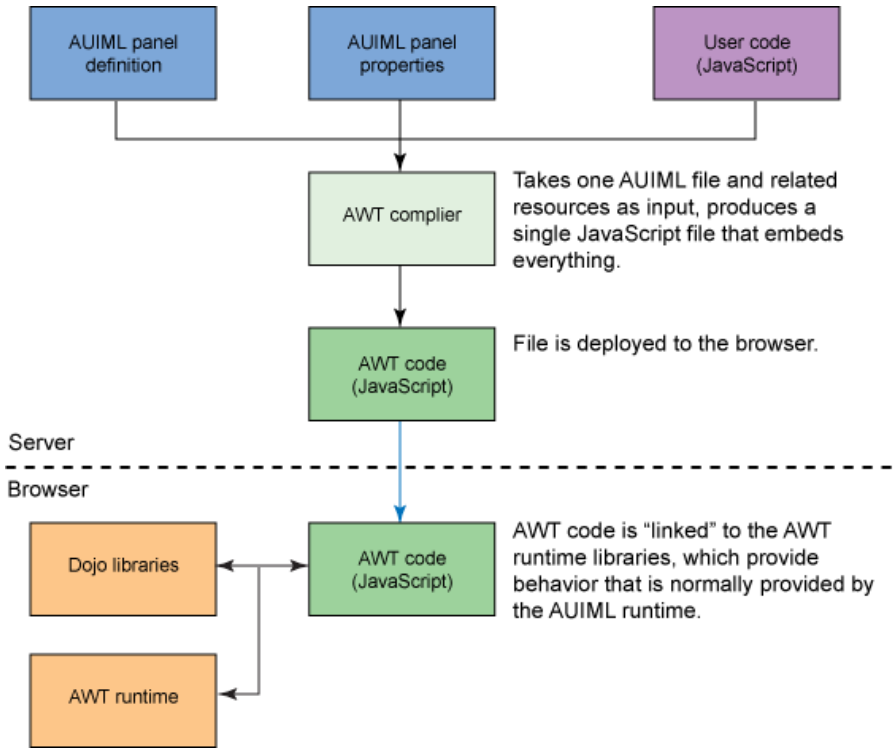

Figure 3.3: AUIML Web Tools architecture [LLPS08]

for AJAX-based web applications. UsiXML is the most universal language, but the 3-layered iterative model transformation is quite complex and requires the use of different tools at different stages. XAML and XUL are the most easy to use solutions, but they are vendor specific. They are also the only ones besides ZUL, which are actually used in real-world applications. AUIML may look promising, but its status is unclear and like RUX it cannot be tested, because the corresponding authoring tools are not yet publicly available.

Summarizing the above, one can say, that at present there is no platform and vendor independent abstract UI description language, which is proven to be capable of producing AJAX-based rich user interfaces. For the sake of correctness, one should bear in mind though, that the small review of technologies above is by no means exhaustive, but rather a collection of some of the most prominent developments in the field.

Table 3.1: Existing abstract UI description languages in comparison

	WebML + RUX	XAML	XUL	ZUL	UsiXML	AUIML
AJAX support	++	+	+	++	+	+
Platform	web	any	web	AJAX	any	?
Vendor	any	Microsoft	Mozilla	ZK	any	any
Models	AUI + CUI	CUI	CUI	CUI	AUI + CUI	CUI
Model generation	from WebML	-	-	-	from CTT	-
Manual effort	+	++	++	++	+	++
Tool support	+?	++	++	++	+	+?
Ease of use	-	++	++	++	-	+
Use in real world	-	++	++	+	-	–

3.2 Model-driven UI development for web 2.0

As discussed in the previous chapters, one of the main benefits of using UI models in web application design is the clean separation between the presentation and the application logic and data structure. Nowhere else this is needed as much as in web 2.0 applications, where the presentation is done by the client and everything else - by a remote server. Apart from the classical client-server architecture, there are also other ones, which benefit from AUI approach. Foremost the service-oriented application architecture, where the client can access multiple, potentially hundreds of web services in different locations. Each single service provides some functionality, which, in the most cases, needs to be reflected in the client's UI. Additionally, each service may be altered, upgraded or even exchanged for another one by its host without notice. In this case a clean, persistent interface to the client becomes vital. And this is precisely what can be achieved with AUIs. The back-end will not even need to know, how the particular front-end UI looks like, on what kind of device it runs, etc.

However, putting the AUI between client and server (or web services) means that all business processes will take place exclusively on server-side. The client will only need to handle the presentation. Although this is exactly what present web applications have all in common, it can have negative consequences in future, when increasing demands towards interactivity raise the question of the client taking greater part in non-persistent data handling.

The AUI itself does not inevitably need to be manually defined. To a great extent it can be derived from different types of descriptions of the server-side application. For example:

- WSDL descriptions of web services;

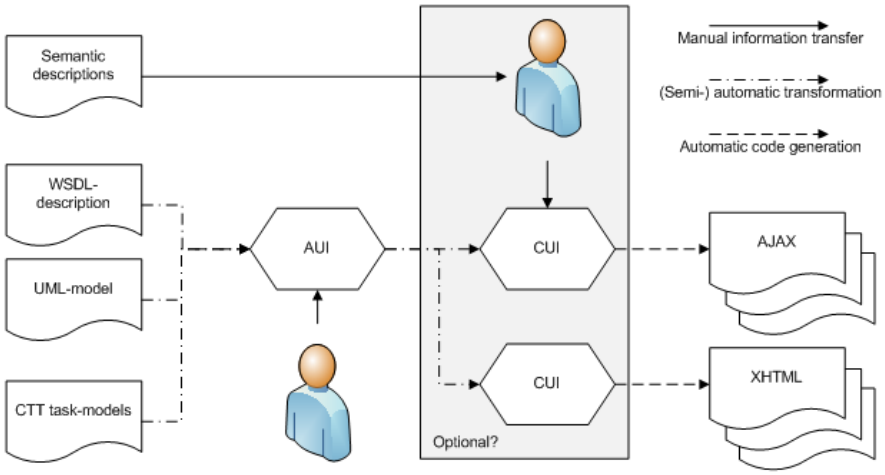

Figure 3.4: General design-flow for model-driven web application development

- Task&Domain models of the server-side application;
- UML-models of the server-side application.

In some cases, generated AUIs may come out large and cumbersome to use, so some small manual enhancements (like a better screen flow) will probably be needed to improve the situation. The overall effort will still be significantly smaller, than when building the AUI from scratch. At least all the UI's important interactive elements can be obtained from the above models, since they all are meant to trigger some functionality on server-side. For example, from a method's expected input data type the exact type of the corresponding input element can be derived (e.g. an integer would require a numerical input field). This is quite simple. More advanced techniques allow one to extract much more information. Although AJAX-applications are often single-page based, this does not mean the extinction of pagination understood as the problem of dividing a complex and big UI into a hierarchy of related elements. And this hierarchy is defined in the AUI too. It can even be automatically generated. A method for automated element arrangement based on CCT task models is presented in [MRVA08].

Of course, some information would still need to be added manually. For example, relations between elements may not be obtainable from the application model.

3.3 AUI and CUI

In the most general approach to model-driven UI development (figure 3.4), the platform independent AUI needs to be refined to the concrete UI before the actual code generation

can take place. In contrast to the AUI, which merely enumerates UI elements and relations, binds them to the underlying data model and, at most, puts them together in groups, the CUI holds a lot more specific information:

- layout (fine-granulated spatial arrangement),
- look&feel (color, size, etc.),
- contents (the actual text, references to images and multimedia),
- behaviors (time or user triggered events)

The CUI is platform dependent. Indeed, you can only define behaviors, which are available on the target platform, etc. Often the content needs to be adapted too: images may need rescaling, media can be recompressed, etc. And of course, the layout and look&feel strongly depend on the target platform's resources. Thus, by creating multiple CUIs one can build different variants of one user interface for different platforms and environments.

The CUI's basic structure can be obtained from the AUI by model-to-model transformation. In fact, many elements of the AUI (like interactive controls) are simply repeated with a number of additional property specifications. In the majority of cases, the new information needs to be added either manually or again by automated extraction from other models. Most of the server-side application models, however, do not hold any information related to the concrete content. In this case, specialized extensions can be used. For example, in the case of WSDL numerous attempts were made to add semantic information to web service descriptions (GUIDD[8], WSDL-S [W3C05b], etc.).

For the manual editing, visual authoring tools can be used. Most of them are quite intuitive and can be operated without any special knowledge about programming, what surely is an advantage. Once the CUI is ready, the code of the final UI can be generated from it fully automatically.

3.4 Code generation from AUI

If the CUI is going to be skipped and code generation is to be done from the AUI directly, the information normally held in the CUI will need to be relocated elsewhere.

- *Layout and look&feel.* In the particular case of web applications, the layout and look&feel information is mainly stored in cascading style sheets (CSS). So the final layout and styling of the UI can be customized independently of the code generation. What CSS currently can't do is pagination, but this is a task of the AUI, not the CUI.

[8] http://inf.josefspillner.de/webservices/guidd.html

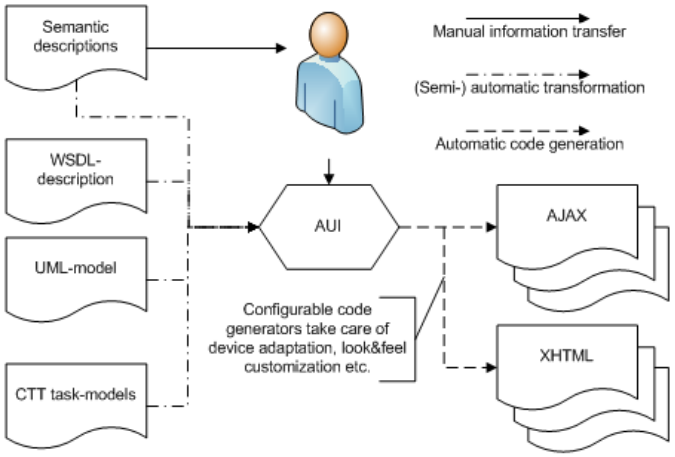

Figure 3.5: Design-flow for web application UI with direct code generation from AUI

- *Contents.* Semantic information can be stored in the AUI directly or via reference to external resources. Of course, the adaption of the contents to the target platform would become a little less straightforward, but not impossible. The code generator can be extended to take care of some of the adaption aspects. Since the code generator is in any case platform aware, options like image resampling and media adaptation can be set by the generator automatically. Other functionality, like choosing the right widget for a certain UI element, may be achieved by making the generator configurable.
- *Behaviors.* The AUI already describes the core functional behavior of every UI element. Of course, there are additional behaviors like roll-overs, animations, etc. They do increase the UI's responsiveness and add some extra feeling of interactivity, but they do not contribute to the interface's functionality. For this reason we shall neglect them in this particular work.

Skipping the mostly manually created CUI would shorten the task queue for UI development (figure 3.5). Only one model would need to be created - the AUI. Of course, it will need to be slightly extended to hold some new information types. This information will also have to be added manually in many cases. On the other hand, the methods of auto-populating CUIs described in section 3.3 are of course applicable for AUIs too. Additionally, having only one model, the problem of inconsistent changes (i.e. applied only to higher-level models) and consistency checking as such will fall away automatically.

Similarly, the code generator will need to be somewhat extended too, but, on the other hand, there will no longer be need for the AUI-to-CUI transformation. The code generator will become the only platform dependent member of the chain. Developing a universal configurable code generator for a certain platform once would enable the programmer to

generate UIs from different AUI descriptions quickly in a single step.

3.5 MARIA XML overview

MARIA XML [PSS09] is a generic platform independent language developed by the HIIS Laboratory of the Institute of Information Science and Technologies (ISTI-C.N.R.) in Pisa, Italy, under the supervision of Fabio Paternò. Based on the experience of TERESA XML, the previous UI description language developed by the group, MARIA XML was made more dynamic and oriented towards user interfaces with a high degree of interactivity, which is exactly what is needed for AJAX applications. At the same time it is still a very technology independent language. This is why it was chosen to be used in this work. The name MARIA stands for Model-bAsed descRiption of Interactive Applications. The language is still being actively developed, so for the scope of this work we shall refer to the pre-release version 1.3.

MARIA XML originally implements the traditional AUI-CUI-FUI approach. It uses different vocabulary to describe the AUIs and CUIs. Since we are particularly interested in direct code generation from an AUI, we will not discuss MARIA XML CUIs here. It should be pointed out, however, that like in other abstract UI description languages, it is the CUI, where information related to styling, spatial presentation, and semantics is to be specified. To skip the CUI, this information will need to be transferred elsewhere.

To start off, however, the main elements of the MARIA XML abstract UI shall be introduced (figure 3.6).

- Presentations are containers with UI elements. Similarly to views or panels in other UI description languages they resemble pages of legacy web applications or windows in desktop applications.
- Connections connect the presentations between each other. There are two types of connections: *simple connections* which merely switch the active presentation and *conditional connections*, which have a customizable behavior. Additionally several connections can be composed to a *complex connection* using Boolean operators.
- Interactors are the actual interface elements. MARIA XML AUI defines several groups of interactors like *only_output*, *control*, *numerical_edit*, etc. depending on their general purpose. Interactors can be grouped together by composition elements (containers): *grouping* and *relation*.
- The underlying application data model can be described in MARIA XML AUI using the XSD type definition language [W3C01]. Instances of the data model can be bound to interactors or used in conditional operations or as arguments to external functions. They can also be handled by the generic script. Data entities are referenced using XPath expressions [W3C99].

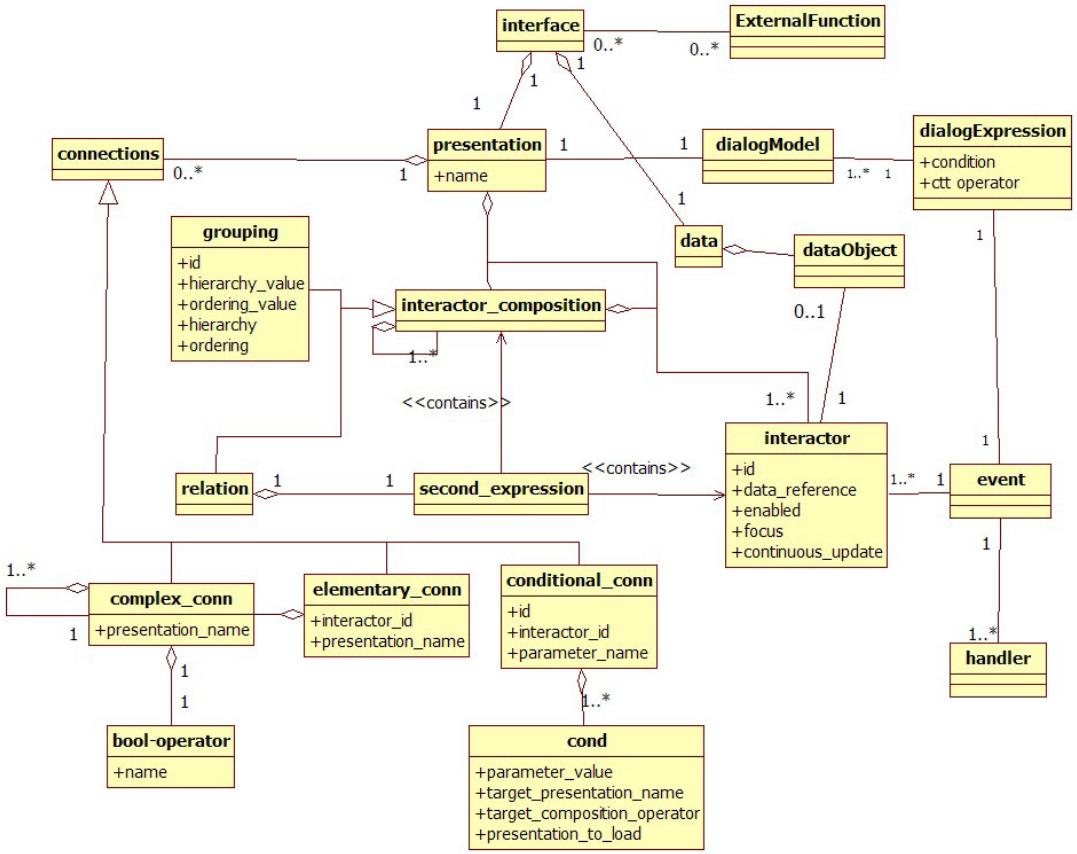

Figure 3.6: MARIA XML AUI overview [PSS09]

- Events can be triggered by most interactors. The event model of MARIA XML AUI is very simple. There are only 3 types of events: *value_change* (e.g. for a text input element), *activation* (e.g. for a button) and *selection_change* (e.g. for select elements or lists). Event handlers can either change interactor attributes in a declarative manner or execute some generic script or raise another event.
- Generic script is potentially a very powerful feature of MARIA XML. The script is thought to be used mainly for event handling. Currently the syntax of the script language is not defined yet. The script will have on-the-fly access to entities of the data model and properties of interactors. Some special script expressions can also be used as dynamic values for interactor properties.
- External functions describe any functionality, which cannot be modeled based on MARIA XML itself. They can define key properties like parameter types and return

values of server-side functions, types of information sources or custom functions of the UI, that need to be implemented separately.

External functions are the gate to the outside world in MARIA XML AUIs. Their concrete implementation will vary depending on the communication technology used. In combination with the data model they build the interface to the back-end-application. By the way, the use of XML Schema data descriptions is very handy when working with web services, because it is also being used in WSDL. So the integration of MARIA XML in service-oriented systems will be particularly easy.

The root node of any MARIA XML AUI is the *interface* tag. It can contain a data model, some external function definitions, and a collection of presentations (figure 3.7).

```
<interface ... >
    <data>
        <schema>
            ...
        </schema>
    </data>
    <externalFunctions>
        <function>
            ...
        </function>
        ...
    </externalFunctions>
    <presentation name="first_presentation">
        ...
    </presentation>
    <presentation name="second_presentation">
        ...
    </presentation>
    ...
</interface>
```

Figure 3.7: MARIA XML AUI code example

3.6 Problem statement summary

As shown in 3.1, abstract UI descriptions based on a single model have attained greater success than the more general ones with two or three levels of modeling. Although a

multilevel architecture clearly offers more flexibility and fine-tuning options, it appears to be too complex. Multiple modeling steps require an entire network of authoring tools and raise consistency issues. The more simple and successful single-model approaches use only platform-specific models without the abstract ones.

The subject of the following chapters is the methodology of generating FUI code from the AUI directly. The abstraction from the underlying platform shall be retained, but only one step in code generation will have to be performed. Thus the advantages of both groups of model-driven approaches will be combined: the simpler and more successful one model based UI generation scheme and the platform independent universal UI model with a high level of abstraction. To keep the possibility to customize and fine-tune generated UIs, a more powerful configurable code generation environment shall be developed.

4 The Concept

The main objective of this work is the evaluation of the possibility of direct code generation from abstract user interfaces and the integration of automated AUI-to-FUI transformation into the development process of AJAX-based web applications. An obvious aim pursued in this case is lowering efforts needed to create a rich UI, based on an abstract description by skipping an entire step (the intermediate CUI model).

In this chapter we shall discuss the generation of code for AJAX-based Web 2.0 applications from MARIA XML abstract UI descriptions in detail. After a general overview of the methodology of direct code generation from the AUI (sec. 4.1), a detailed discussion of each major element of the AUI will follow (sec. 4.2). Possibilities to link abstract UI elements to concrete widgets and AJAX constructs will be evaluated.

In section 4.3 some extensions to MARIA XML AUI will be proposed. Finally, the code generation template structure and ways to customize the resulting application will be presented in section 4.4.

4.1 Model-driven UI generation

First of all, the most important instruments for UI code generation from an AUI description need to be introduced (figure 4.1). The core component is undoubtedly the code generator tool. A typical generator expects the following inputs.

- Metamodel. In our case it will be the model of MARIA XML AUI. It is neither application nor platform specific and always remains the same.
- Model. This is the MARIA XML AUI description of the desired UI. It is, of course, application specific, but should remain platform independent.
- Code generation template. Ideally, there is one template per target platform. To fit into the above categorization, it can be described as platform dependent, but not application specific.

Code generation templates are written in special languages and describe how to transform a model into program code. Depending on the quality and versatility of the template, the generator can produce code for a specified platform for any application using a single

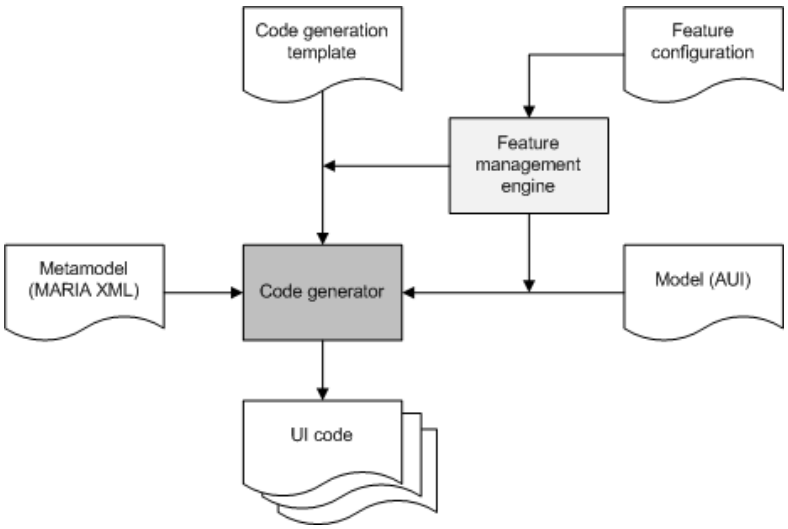

Figure 4.1: UI code generation overview

code generation template. On the other hand, one template, of course, always produces the same result for a specified model. Should the result need to be changed, either a new template has to be created or the existing template has to be altered. In real product line engineering, however, it is often necessary to quickly create variants of a product. To cope with this problem, many code generation platforms support feature-based configuration of code generation templates and models. In this way, models and templates can be non-invasively changed depending on some configuration before actual code generation (similarly to aspect oriented programming techniques).

In the particular case of direct code generation from AUIs, especially numerous customization possibilities are left to the generator. In the vast majority of cases, elements of the AUI can be implemented in different ways. Using feature control, it is possible to switch quickly between predefined implementation variants. It does not affect the template itself, it rather gives the developer the possibility to tweak (advise) it selectively, leaving the main structure intact. So, instead of manually customizing every UI variant with the help of CUIs, we can use configuration files to quickly change the outcome.

4.2 MARIA XML AUIs

The following subsections will present each of the major components of MARIA XML AUIs and identify elements of AJAX-applications that can be generated from them.

4.2.1 Presentations

Each presentation has a unique identifier and may contain interactors or interactor groups and a list of connections to other presentations. The structure of the presentation data type is shown in figure 4.2. It makes sense to populate a presentation with logically connected elements.

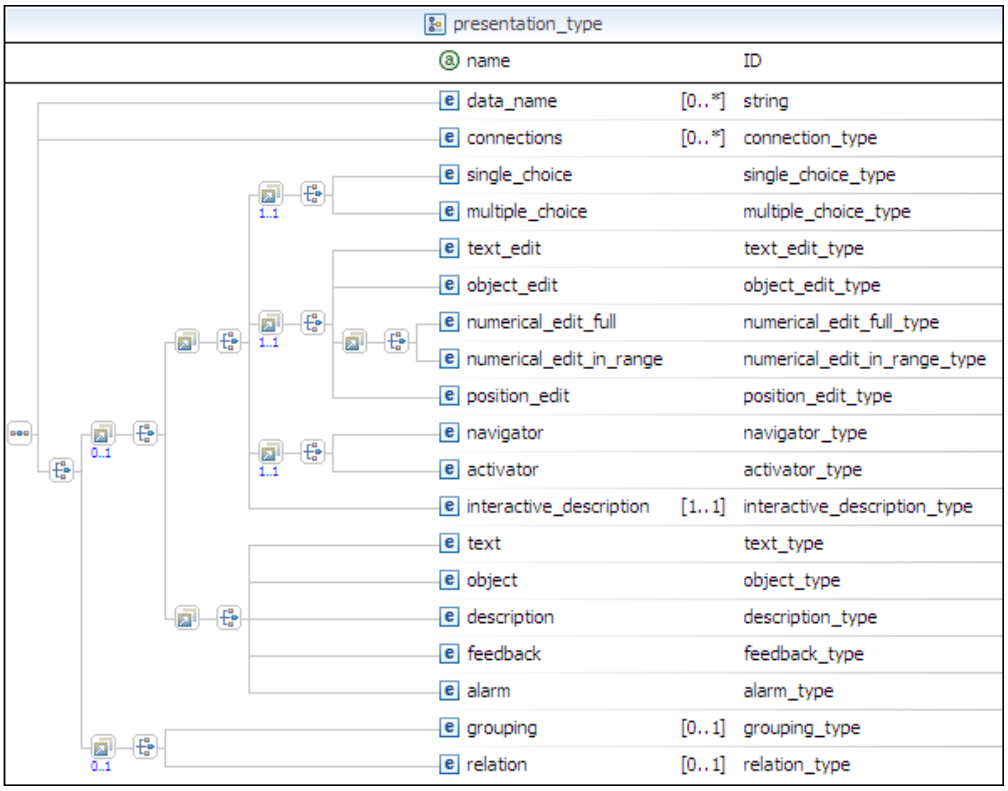

Figure 4.2: MARIA XML presentation element

The atomic UI elements, that we will discuss later, can be grouped together either in *grouping* or in *relation* elements. The former is a simple group of interactors, while the latter assumes a functional relation between them (for example, similar to an input form). Both types of composition elements may hold other grouping and relations.

When generating code for AJAX-based applications it makes sense to create a neutral container element (like) for every grouping and a <form> element for every relation. Both types of interactors have unique identifier attributes in MARIA XML. These IDs can be directly used as XHTML *id* tag properties in order to make the elements accessible using the *getElementById()*-method in JavaScript later.

The result of such a transformation will be a hierarchy of and <form> tags for every presentation, which is perfectly good to use with CSS. It makes sense to save every presentation as a separate XHTML-document, so they can be dynamically loaded upon request.

However, they are not yet complete XHTML-documents - the header and the enclosing <html> and <body> tags are missing. There are two ways to include these elements:

1. The XHTML-header can be saved with every presentation;
2. A single template document containing the header can be defined for the entire application and the presentations would then need to be loaded into the template's <body> section or any lower-level container.

Choosing the first possibility would make loading the presentation very simple: it would be accessible by an ordinary HTML-link. One of the key features of AJAX is, however, the partial content refresh, which could mean loading a presentation into a part of another one. The header of the loaded presentation would need to be stripped at runtime. On the other hand, creating a static header for the entire application, would mean loading all presentations dynamically. In this case, the resulting application will be truly single-page-based. However, customizing the now static header for different parts of the application may become difficult, but this is a common issue of AJAX-applications.

In the following only the second option will be regarded because it fits the general concept of an AJAX application very well. On application startup, the root presentation will be loaded, which is the first presentation in the model.

There is also a possibility to mix both approaches. However, in this case one would need to distinguish between standalone presentations and containment presentation on model level. This would require much more careful planning, as every containment would always require a suitable container. A more complicated model structure could also make the generation of the AUI more difficult. The flexibility of such an approach makes it, however, an interesting subject for a future study.

4.2.2 Connections

The three types of connections in MARIA XML AUI (figure 4.3) are listed below.

1. Elementary connections. These just replace the current presentation by another one. An interactor needs to be specified as a trigger.
2. Conditional connections. These connections can define multiple actions, which can be executed at runtime depending on a parameter. They too are triggered by an interactor, but the target presentation (specified in the attribute

presentation_to_load) needs to be loaded into a *grouping* or *relation* (the *target_composition_operator* attribute) of the current one, thus allowing to refresh only parts of the UI. The parameter can either be a property of an interactor or a variable value defined in the data model.

3. Complex connections. A complex connection is simply a container, which can hold multiple elementary and/or conditional connections bound by Boolean operators.

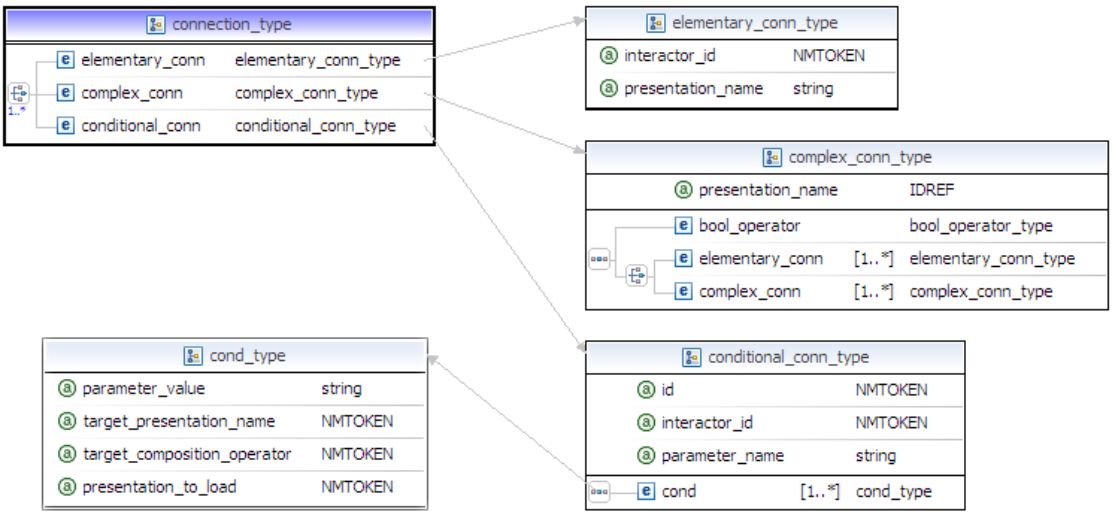

Figure 4.3: MARIA XML connections

For an AJAX-application partial content refreshes, achievable by the use of conditional connections, are of special interest. The implementation is actually straightforward: the desired presentation is requested from the server and placed inside a grouping or a relation. Since these are transformed to and <form> tags, respectively, as explained in section 4.2.1, the server's response can be simply put into the *innerHTML* attribute of the corresponding DOM-element or equivalent properties of custom widgets if used.

If no *target_composition_operator* is defined, the presentation will be loaded to the document's root element (typically the <body> tag). Elementary connections can actually be treated as conditional connections with no *target_composition_operator* and an always valid condition (like "*if (true)*").

Of course, if a multi-page application is desired, simple connections or conditional connections with no *target_composition_operator* can be used to load an entirely new page.

4.2.3 Interactors

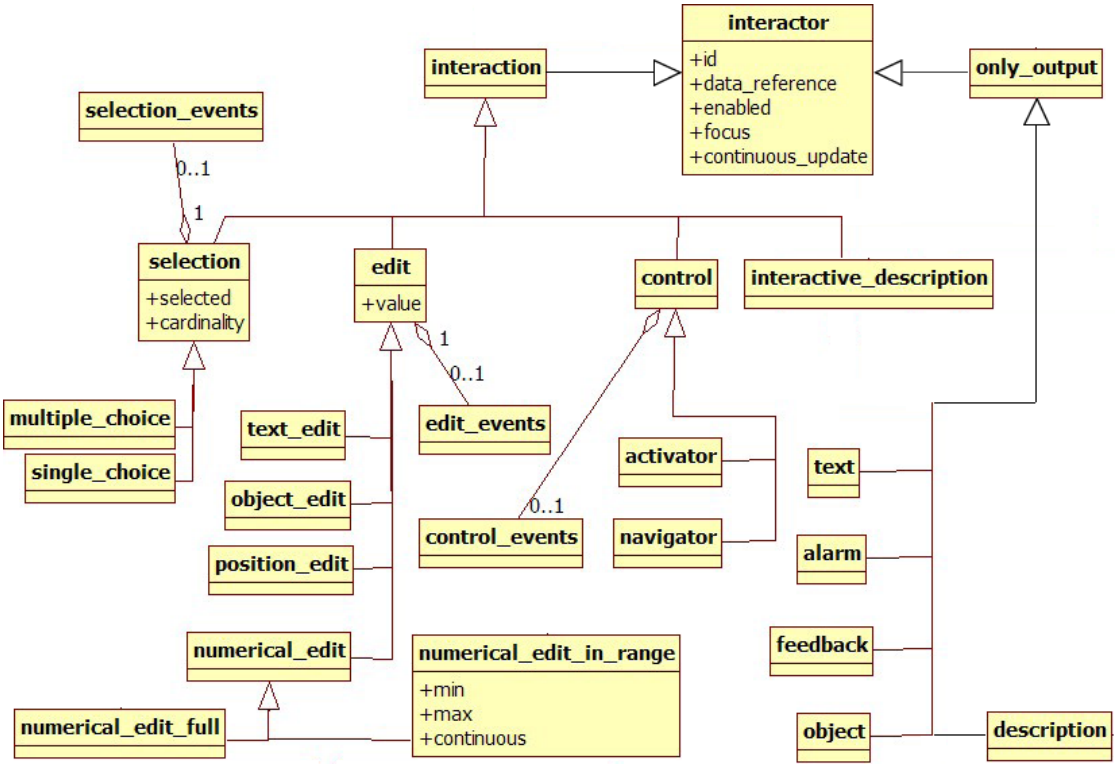

Figure 4.4: MARIA XML interactors overview [PSS09]

In MARIA XML AUI two main categories of interactors are defined: *only_output* and *interaction* (figure 4.4). Obviously, the former can be represented by non-interactive UI elements, while the latter should be implemented with form controls or custom interactive widgets. Some of the most important examples of AUI interactors and corresponding (X)HTML implementations are listed in tables 4.1 and 4.2.

Let us first have a look at *only_output* interactors. Text is the simplest type of content and does not need any special attention. The *object* interactor requires more careful handling. It can hold images, multimedia or any other type of non-textual content. Accordingly, there are several ways to display this content: an image (a thumbnail in case the actual source is of another kind), a downloadable file, or embedded media (audio, video, flash, etc.). Choosing the desired option can be organized in different ways:

1. It can be left to the underlying platform (a JavaScript function can make the decision based upon the source type);
2. It can be made a configuration option for the generator;

Table 4.1: MARIA XML *only_output* interactors

MARIA XML interactor	XHTML/JavaScript element
text	<p>
object	<img>
	link to downloadable file
	embedded media
description	<p> or
feedback	<p>, or pop-up message
alarm	embedded audio

3. It can be defined in the model by the designer.

Options 1 and 2, however, imply a fixed behavior for all objects, which is generally not a good solution. Of course, the generator or the application script could be configured to treat images and video sources in different ways. However, if the designer wanted to make a particular image downloadable by defining a download link next to the image itself, this approach would not suffice. To handle such cases, it is necessary to introduce a way to specify the expected display type of the object in the model already, so that two *object* interactors can be added with the same source, but different display types. The appropriate extensions to the AUI will be introduced in section 4.3.1. For further customization the generator's configuration can be used to override display types for certain media formats.

The remaining interactor types *description*, *feedback*, and *alarm* can be used for different purposes. The code generated from them can vary accordingly. For example, descriptions and feedback can be just elements (perhaps with a special style) and an alarm can be implemented, for instance, either as an embedded audio signal or a flashing graphic.

Table 4.2: MARIA XML *interaction* interactors

MARIA XML interactor	XHTML/JavaScript element
single_choice	<select>
	radio buttons
multiple_choice	<select multiple="multiple">
	check boxes
text_edit	<input>
	<textarea>
numerical_edit_full	<input> with type checking
numerical_edit_in_range	number spinner
	slider
object_edit	image map
	upload control
	flash
navigator	button
activator	button

For interaction elements, there are also multiple ways to implement an abstract interactor in most cases. The list of possibilities presented in table 4.2 represents just the most simple and straightforward implementations. There are a great number of JavaScript toolkits, however, that provide widgets of all kinds. So, how to select the right option for a specific project? The easiest way is to hard code the corresponding mappings in the code generator templates. Switching from a <select> element to a set of radio buttons in the case of the *single_choice* interactor would mean changing the template or just using a different one.

A more elegant solution can be provided by the use of aspect oriented programming techniques. If the main code generation template was built cleverly enough, advices for every interactor type can be provided. In this case, simply changing the configuration options of the generator will be enough to switch from one predefined mapping to another. This is a very powerful approach, which will be discussed later in greater detail.

In addition to direct interactor-to-widget mappings, properties of data model entities connected to interactors can be used to derive the most appropriate widget type (see section 4.2.4). For example, there is no special interactor for date input, but if the interactor is bound to an element of the data model with corresponding type restrictions, then the generator can assume, that a calendar widget is applicable.

Every interactor has a set of attributes, which define its properties. Figure 4.5 shows the *numerical_edit_in_range* interactor as an example. The first group is the standard set of attributes for interaction elements. Then special attributes for this concrete interactor

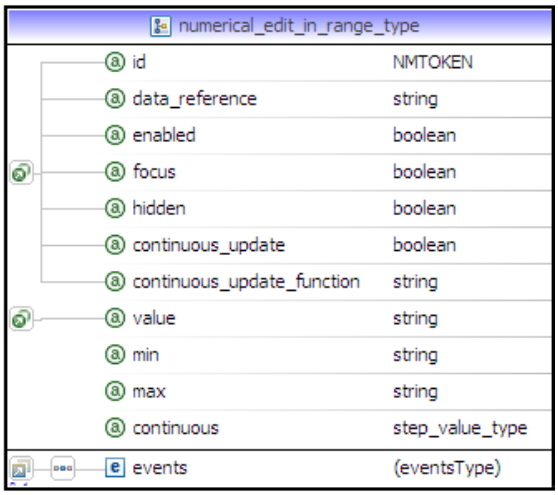

Figure 4.5: MARIA XML *numerical_edit_in_range* interactor

type follow: the initial value, upper and lower borders, and the step size. At the very bottom you can see container element for a list of event handlers. They will be discussed in section 4.2.5.

All of these attributes, except for the *id*, need to be accessible at runtime. Event handlers and generic script (section 4.2.6) can read and change these attributes. There are two ways to implement this:

1. Map the abstract attributes of interactors to concrete attributes of the corresponding widgets during code generation;
2. Provide JavaScript wrapper functions to set the concrete UI element's properties according to its abstract prototype's attributes at runtime.

The first option saves resources at runtime as no additional transformation is needed, but requires a more sophisticated code generator with a good set of string processing functions. For example, XSLT - an obvious candidate for the transformation of an XML-based AUI to XHTML code - only supports string transformations from version 2.0 [W3C07] onwards. Should the desired code generator language be incapable of such transformations, there is always the possibility to fall back to option 2, but it will imply more calculations at runtime and, thus, a potentially slower resulting application.

4.2.4 Data model

In the particular case of AJAX applications, non-persistent UI data is normally stored in JavaScript variables. It makes sense to instantiate such a variable for every element

of the MARIA XML data model. The concrete implementation of the tree-based data model in JavaScript is unimportant. It is essential, however, that XPath-like queries can be performed efficiently.

The XSD data type information in the model can be used for on-the-fly validation. However, since JavaScript is a loosely typed language, all the type validation functionality needs to be explicitly provided by the programmer. So it will be up to the code generation template developer to include type checking mechanisms. As mentioned before, if a JavaScript framework is used, which offers special widgets like number spinners, calendars, etc., data type information can be used to select the most suitable widget. Restrictions like minimum and maximum values or string patterns are of special interest too. Many widgets already support the definition of such constraints. If not, custom validator functions will need to be added. These functions should be executed every time an action is performed upon the element of the data model. In general, data types used in the AUI should mimic the expected data types of the server-side application as close as possible to provide maximum consistency.

The data references in the AUI (similarly to the interactor's attributes) can be processed either during code generation or at runtime. Which option to choose, again, largely depends on the code generator used. If the generator is capable of transforming the XPath notation to the one appropriate for the concrete implementation, than, perhaps, transforming the references during code generation is the better choice. If not, the XPath references can also be left in place and processed by JavaScript at runtime each time a reference is used (or at least at application startup). This will, certainly, require additional resources at client-side and the performance of applications with a complex data model or large amount of bindings will drop.

The links between the data model and the interactors present another challenge: every time an interactor is used, the data bound to it must be updated immediately and vice versa. In the first case, DOM-events like *onChange* or *onClick* can be employed, but in the second - there is no such ready-to-use mechanism. A way to cope with this problem is to use setter-functions to access the variables. Such a function can easily check for links to interactors and perform the appropriate updates after saving the actual data.

If the generated application is single-page-based, the data will automatically remain intact during the application's runtime. In the case of a multi-page application, this may introduce additional challenges. All set variables will need to be transferred from one page to another. On the other hand, if many presentations are loaded and removed dynamically, one should take care of orphan data and references to avoid memory leaks.

4.2.5 Event model

MARIA XML AUI has a very simple event model with only three events.

Activation events are triggered by activator and navigator interactors. They can be mapped to *onClick* DOM-events of the corresponding buttons or other UI controls.

SelectionChange events are fired when the selected element of a single or multiple choice interactor changes. The DOM defines the *onChange* event for this purpose.

ValueChange events indicate that the value of an interactor has changed. Depending on the widget used, these events can be mapped to *onKeyUp* or *onChange* or other similar DOM-events.

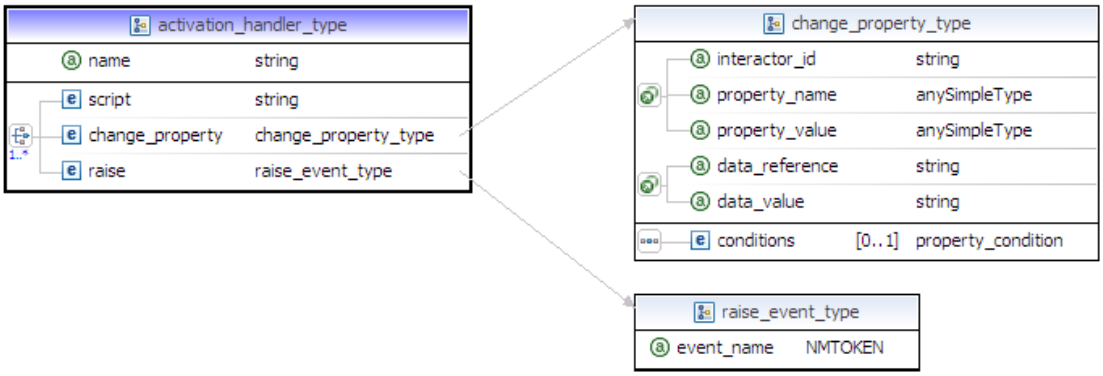

Figure 4.6: MARIA XML event handlers

There are three ways to handle these events. One can either execute a script (for more information about scripts see section 4.2.6), or set an interactor's attribute or data element to a specified value, or raise another event. Any number of each of these actions can be defined in the event's handler list (figure 4.6). For the *change_property* handler conditions can be defined based on the values of other interactor's properties or data model elements. Figure 4.7 shows a code excerpt from a MARIA XML AUI with a simple example: a login box is shown or hidden depending on the user indicating if he had already registered or not. It could be, for example, part of a sales order form with optional registration.

When generating an AJAX-application, it makes sense to create a JavaScript function for each event handler. This function will be triggered by a DOM event fired by the corresponding element as described above. For script-based handlers the function will basically only need to contain the eventually preprocessed script (section 4.2.6). JavaScript expressions setting DOM element properties will need to be generated from *change-property* event handlers. Event raising handlers will merely need to call another event function.

Although the three built-in events of MARIA XML AUI will probably be enough to provide general functionality of the UI (navigation and button-triggered actions), there

```
<handler>
    <change_property
    interactor_id="loginRegistered"
    property_name="hidden" property_value="false">
        <conditions>
            <condition
            interactor_id="registeredBefore"
            property_name="selected" property_value="Yes"/>
        </conditions>
    </change_property>
</handler>
<handler>
    <change_property
    interactor_id="loginRegistered"
    property_name="hidden" property_value="true">
        <conditions>
            <condition
            interactor_id="registeredBefore"
            property_name="selected" property_value="Yes"/>
        </conditions>
    </change_property>
</handler>
```

Figure 4.7: MARIA XML AUI event handler code example

certainly are more events to choose from in an AJAX-based application. Typical examples are mouse, keyboard, touch or time triggered events. On the other hand, however, these advanced events are much more platform specific, than the basic ones. To keep the model platform independent we will refrain from adding other events to the AUI, but rather take a closer look at ways to customize the concrete implementations of the three basic event types at the code generator level. Firstly, the type of the element, which fires the event, should be taken into account to determine the concrete mapping to DOM-events. For example, the *value_change* event can be mapped to the *onKeyUp* or *onBlur* DOM-event in the case of a text input field, while for a selection element using the *onChange* event makes more sense. Secondly, the default event mapping can be customized by using the code generator's feature control.

4.2.6 Generic script

The present version of MARIA XML (1.3) does not specify any exact script syntax. However, it comes with some examples utilizing the script, part of which you can see in figure 4.8. The code excerpt shows an event handler, which checks the validity of the entered user name and password with the help of an external function and, depending on the result, sets a session variable's value in the data model and shows or hides an invalid-login warning. Note, that the session variable is just named in this way. The lifetime of variables generated from the MARIA XML data model is limited to the user session in any case, because they are only stored on client-side.

```
<handler>
    <script>
        <![CDATA[
        if(external:login(data:Login_entry/login,
                    data:Login_entry/password)
        {
            data:UserSession/loggedIn = true;
            ui:InvalidLogin.hidden = true;
        }
        else
        {
            data:UserSession/loggedIn = false;
            ui:InvalidLogin.hidden = false;
        }
        ]]>
    </script>
</handler>
```

Figure 4.8: MARIA XML generic script example

The type of the expression used is specified by a prefix, separated by a colon. There are three types of elements, which need to be accessible by the script.

1. Data model elements are currently referenced by the prefix *data:* followed by an XPath expression (e.g. *data:userSession/loggedIn*).
2. UI element's attributes are referenced by the prefix ui: followed by the interactor's id and the desired attribute separated by a dot (e.g. *ui: loginButton.hidden*).
3. External functions (section 4.2.7 for details) are referenced by the prefix *external:* followed by the function name (e.g. *external:validateLogin()*).

There is also the construction *exp[...]*, which indicates that the contained string needs to be evaluated as a generic script statement. Such expressions can be used as values of attributes of interactors.

Similar to the data model and interactor references, the generic script can either be transformed to a platform specific script language (JavaScript in the case of AJAX) during code generation or interpreted by the platform's script language at runtime. In either case the required efforts cannot be reliably estimated without a clear specification of the abstract script.

For the scope of this particular work the script's syntax shall be assumed to be identical to JavaScript. Although JavaScript is clearly platform specific, it's syntax can be used in an abstract script just as well as any other syntax. Using JavaScript will clearly benefit the generation of AJAX-applications, but it will not have any significant impact on other platforms. Target platforms without explicit JavaScript support will need to interpret it just as if it was a special script language for MARIA XML.

The prefix-based references described above will not fit in the JavaScript syntax, however. To cope with this problem, we can rewrite them a little without any impact upon the semantics. Table 4.3 shows an example set of substitution rules. Actually, they can also be executed automatically before evaluating the abstract script if the original notation is to be kept in place.

Table 4.3: MARIA XML generic script to JavaScript transformation

MARIA XML generic script	JavaScript substitution
data:xxx/yy = zzz;	setData('xxx/yy', zzz);
data:xxx/yy	getData('xxx/yy')
ui:xxx.yy = zzz;	setUi('xxx.yy', zzz);
ui:xxx.yy	getUi('xxx.yy')
external:xxx(...)	externalFunctionXxx(...)

In a sense, the above set of rules represents a simple script translator already. With more differences between the abstract and the concrete scripting languages its complexity will rise. This is not much of a problem if the transformation is handled by the code generator, but if it is done at runtime, long scripts may need considerable time to be evaluated. Bearing this in mind, the absence of a fixed syntax for the abstract script may actually be an advantage. Every developer can decide for himself, what scripting language is appropriate for the concrete project's specific needs. The only requirement would then be the adaption of the three special expressions (references) above to the chosen syntax before actual execution.

4.2.7 External functions

External function model entities hold information about the names and data types of function parameters. There can be any number of input parameters, but there has to be only one return value. An example of an external function is shown in figure 4.9.

```
<function name="login" type="web_service">
  <output type="xs:boolean"/>
  <input name="login" type="xs:string"/>
  <input name="password" type="xs:string"/>
</function>
```

Figure 4.9: MARIA XML external function example

There are four different built-in types of external functions in MARIA XML AUI.

webService. This type of external function is supposed to be used to communicate with a web service or a script-based web page.

database. This type of function is expected to return the result of a database query.

feedRss. This function can be used to query a feed-based resource.

code. This can be any custom code executed at client side. The actual code will need to be added to the generated application manually.

In the special case of an AJAX application the first three types of functions actually require a request to be sent to a server and should return the result in some predefined form. Following the general idea of a seamless interface flow, it makes sense to implement such functions using the XMLHttpRequest-mechanism. The server address can either be hard coded in the code generator or passed to it as a configuration parameter, or passed to the function itself as one of its own input parameters. In the latter case, the model designer will need to know the exact URIs. On the other hand, if the URI is set in the generator, the generator will become application-specific, which means it (or at least its configuration) will need to be changed for every application. Perhaps, both approaches can be beneficial depending on the concrete project. One should also keep in mind, that there is a limit to the length of GET-parameters. For larger data sets, the POST-method is the better choice.

Since JavaScript lacks a facility to query a remote database by itself, there will be a server-side script needed to actually execute the query. There are endless possibilities to implement such an external function: the query can be built on client-side and sent to the server to be executed, or the client may only supply a number of parameters to the server script to build the corresponding query, or any combination of the two.

The fourth external function type - *code* - can be used to generate a JavaScript function template, which instantiates an empty function with the required parameters and, perhaps, some textual description (e.g. JavaDoc). As mentioned before, the actual code will need to be added afterwards. Of course, similar parts of the server-side applications can be generated too if the UI is not intended to be used with existing ones. For example, an external function of the type *webService* can be used to generate a WSDL-description for a web service or a server-side script with function templates.

4.2.8 Continuous updates

MARIA XML explicitly supports the possibility of continuous field updates, which can be used to describe behind-the-scene communication between client and server in an AJAX-application. Most of the MARIA AUI interactors have special attributes: *continuous_update* and *continuous_update_function*. The former is just a Boolean switch to indicate, whether a continuous update of the element is desired, while the latter can be used to specify an external function to handle the background communication.

Generally, continuous element updates can be organized in several ways in an AJAX application.

1. Polling, when the client continuously sends update requests to the server. This technique has two subtypes:
   a) Normal polling, when the browser permanently queries the server and receives immediate responses and
   b) Long polling, which is based on long-lived connections and server responses, which are sent only if a new event occurs.
2. Streaming, when the client establishes a connection and listens for update messages from the server.

Each approach is beneficial and worth to be supported in certain cases. From the point of view of an abstract UI description there is no difference, as the underlying logic can be easily encapsulated in an external function in either case. Supporting some special switch in the platform independent AUI model does not make any sense, because the choice mainly depends on the platform used. So the choice of the continuous update technique shall be left to the code generation template developer. It can also be made feature-controlled.

4.3 Extensions to MARIA XML

In section 3.3 it was mentioned already, that in the case of a 2-step generation approach, the CUI holds all semantic and layout information. Should the CUI be dropped, this information needs to be included in the AUI already. Some extensions to MARIA XML AUI addressing this issue will be proposed in this section. Although these extensions were elaborated based on the example of AJAX application generation, great care was taken to keep the AUI as platform independent as possible. All the introduced features can also be used for other platforms in one way or another.

4.3.1 Styling information in the AUI

In the special case of web applications, most of the layout information can be outhoused to the CSS style sheet, but references to the corresponding CSS classes should be included in the AUI. The easiest way to do this is to add an extra string attribute (*style_class*) to each and every interactor, where the id of a style definition (e.g. CSS class) can be saved to. Figures 4.10 and 4.11 show the *text* and *single_choice* interactors extended by this attribute. Styling attributes can also remain empty, of course, if no special appearance is desired for the UI element. Style-based look&feel control is not exclusive to web applications. Most programming frameworks support similar concepts (like Java Swing look&feels for example), so using references to visual styles in the model does not make it seriously platform dependent. The corresponding new attribute will only contain the id of the style. The platform-specific code generation template will define how exactly it is to be used.

The concept of UI skins also enjoys rising popularity among web developers. Many JavaScript frameworks support CSS-based skins. The crucial difference to the traditional styling is that the skin is set once for the entire application and there is no need to specify the applicable style for each element. To support this paradigm, the *style_class* attribute shall also be added to the top node of the AUI model - the *interface*.

As discussed in section 4.2.3, the *object* interactor can be handled in multiple ways. In order to give the designer more control over how a particular *object* is displayed, the *display_type* attribute is introduced, which will take one of the following three values.

Image This display type shall transform any kind of media to images. These can be thumbnails or media-type icons or anything else depending on the concrete implementation.

Media Objects with this display type shall be shown as embedded media if supported by the underlying platform. If no support is given, it is up to the developer to provide fallback functions.

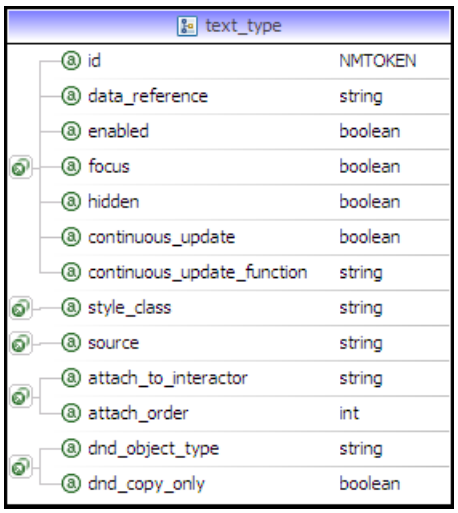

Figure 4.10: Extended *text* interactor

Save This last display type shall offer the end-user to save the object's content to a user-defined location. In the case of web applications this corresponds to an ordinary download.

Similar differentiation needs to be undertaken for the *text_edit* interactor. Not just in web applications, but about everywhere, where text input is expected, one can distinguish between three text-input display types enlisted below.

Short text can be edited in simple in-line input fields. Generally it does not require any special formatting or handling.

Long texts may require some formatting and can have multiple lines. To edit these texts multi-line input fields are used. Sometimes, input widgets also provide basic or even advanced formatting functionality (i.e. WYSIWYG editors).

Secret texts should not be shown at all. Commonly secret input is replaced by asterisks on-the-fly. This type of input is generally used for passwords, admission-keys etc.

Thus, a *display_type* attribute taking these three values will be added to the *text_edit* interactor too.

4.3.2 Adding the content

There are UI elements, which are intended to contain some meaningful information by themselves. Foremost, the text output elements, but also other types of output like

images and multimedia. Selection elements also have some content information, namely a list of available choices.

To add content information to the AUI several approaches can be used. The simplest way would be adding some new attributes and filling them directly with texts, image sources, name-value pairs for selection choices, etc. This is very straightforward, but at the same time very inflexible. What happens if the resulting application should be multilingual, for example? In this case, there should be a possibility to switch between content sources at runtime. Here references to some external sources for the content can be used. In the simplest case, the reference can be made static and the source will have to be relied on to supply the correct version of the content, based on some inner-application indicators. Another possible solution is making the references themselves depend on variables from the data model. This can be done by using a script-based attribute value like *exp["someRef"+data:lang]*. In this example the computed reference will depend on the data model element *lang*, which can hold, for instance, a language id.

Many more referencing mechanisms can be thought of, but from the point of view of extending the AUI, the essence is always the same: a new string attribute is needed to attach the reference to the interactor. We will call it *source*. It can either hold the contents directly or by reference. Which option to choose, does not need to be specified in the metamodel, but rather can be left for the developer to decide based on his current needs and/or the capabilities of the used code generation engine. Figures 4.10 and 4.11 show the extended *text* and *single_choice* interactors.

In the special case of choice options for selection elements, there is another possibility to specify the content. A new model entity - the *choice_option* - will be added with two attributes: an *id* and a *value* (figure 4.11). Once the choice options are included in the model itself, the generator will be able to iterate through them easily without having the need to know the concrete formatting of an external source, etc. To give the developer as many options as possible, however, the possibility to specify a single external source for the selection options (the new *source* attribute) will be left in place too. In this way, either external or internal sources may be used in each case.

4.3.3 Semantic interconnections between UI elements

In certain cases the main purpose of a UI element is to describe another one. Common examples are tooltips, labels, floating bubbles etc. To model these types of elements, there is need for a possibility to establish a connection between a describing interactor and the described one.

To cope with this problem, the string attribute *attach_to_interactor* is added to each interactor of the only_output group (note the corresponding attribute group in figure 4.10). This attribute will hold the id of the described interactor. That is, if the designer

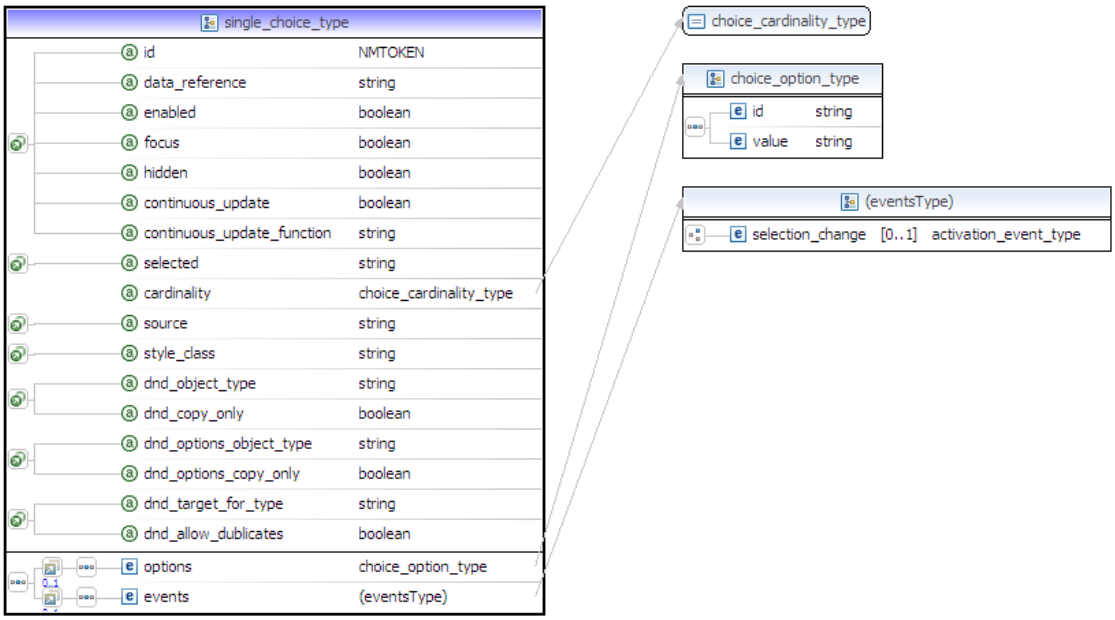

Figure 4.11: Extended *single_choice* interactor

would like to define a textual label, he would need to create a new *text* interactor and specify the id of the interactor to be labeled in its *attach_to_interactor* attribute.

To support more than one description per element and the possibility to compose complex descriptions, another attribute is introduced: *attach_order*. Its value can be positive or negative integers and will by default be 0. The concrete interpretation of this attribute will, of course, depend on the code generator. A simple example would be to place describing interactors with negative *attach_order* values before the described element and the ones with positive values - after it. Alternatively, a value group can be interpreted as floating labels for instance. The absolute value of the attribute can be used to specify the exact composition order if multiple interactors of the same type are attached to a single target.

4.3.4 Drag and drop

Perhaps the most prominent example of rich interactions, which are considered a key feature of Web 2.0 applications, is the drag-and-drop (drag&drop) functionality. In fact, “rich interactions” is a very popular term, which vaguely defines interaction types, which are typical for desktop applications, but cannot be achieved using the standard set of HTML UI controls. Indeed, drag&drop surely is very popular in desktop UIs, but it is

still very rare on the web. However, with the help of JavaScript, drag&drop can also be used in web applications.

In contrast to other rich interactions related to instant reactions to user input or partial UI updates, drag&drop cannot be described by MARIA XML AUI by default. To provide this possibility, the following attributes are proposed to identify an interactor as a draggable object:

dnd_object_type to set a string type identifier in order to be able to specify a type-specific drop zone;

dnd_options_object_type to specify a mutual object type for all options of a selection element;

dnd_copy_only to specify, if the object itself or merely it's copy is to be moved;

dnd_options_copy_only to set the *dnd_copy_only* attribute for all available options of a selection element.

Any interactor can be made draggable. On the other hand, only interactors, capable of containing others, can be used as drop targets. These are *grouping*, *relation*, *single_choice*, and *multiple_choice*. These interactors are additionally extended by the following attributes to identify them as possible drag&drop targets:

dnd_target_for_type to specify the acceptable object type;

dnd_allow_duplicates to specify, if an object can be dropped more than once.

How exactly an object is handled when moved or dropped onto a target, shall be left for the code generation template designer to decide. Few available JavaScript widgets support drag&drop. However, in the most cases, a simple fallback based on basic UI controls can be provided. For example, each draggable interactor of a certain *dnd_object_type* can be provided with a button for each of the accepting targets. Pressing this button would virtually move the object to the target.

4.4 The code generator

In this section code generation templates will be discussed in greater detail. The chosen code generator framework will be assumed to support feature management or any other possibility to customize the output. If feature management is not explicitly supported, it can be substituted by generating the code generation templates themselves from a generic template and some sort of configuration file.

Note, that in contrast to the previous section, the discussion will be limited to AJAX code generation exclusively because code generation issues are largely platform specific, as opposed to abstract descriptions. The approach, proposed below, can, of course, be used to generate any kind of code in any programming language with similar considerations.

4.4.1 Code generation templates

An AJAX-application generally consists of HTML or XHTML and JavaScript code. Whether to use embedded JavaScript or a separate extension file shall be left to the template developer. Putting all the JavaScript in one file will make it more redundant as not all the script (i.e. not for all possible presentations) will be needed each time. On the other hand, the file can then be cached by the client, thus reducing bandwidth usage. Using embedded JavaScript makes the client load parts of script from the server with every presentation. Additionally, if partial UI updates are used (and this will be mostly the case), the embedded script will need to be evaluated explicitly immediately after being loaded.

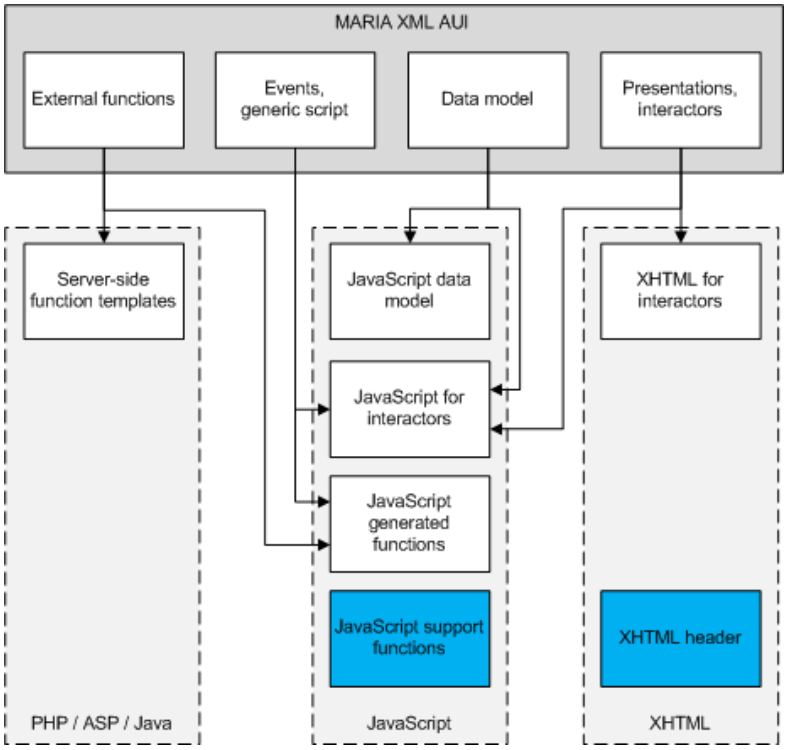

Figure 4.12: Code generation template structure

Figure 4.12 shows an overview of AJAX code generation from MARIA XML AUIs. Note the two highlighted elements: these parts of the final application will need to be written manually. They cannot be generated. The JavaScript support functions represent the code, that will control widget properties, take care of loading new presentations, reference management, parsing and evaluating generic script, and data bindings (if needed), etc. Generating server-side functions is optional. As stated before, it makes only sense if the

server-side application is yet to be developed.

Interactors are represented by (X)HTML elements or form controls as discussed in section 4.2.3. However, in order to provide data binding and event-based functionality, most of them will generate JavaScript functions too. Additionally, many special widgets have to be instantiated by JavaScript and then placed somewhere in the DOM. Interactor-related functions can also be influenced by the data model (i.e. for data type checking, widget selection, etc.).

4.4.2 Customizing the generator

Giving the code generation templates a fine-granulated structure is essential if feature management is to be used. Using it, the designer can replace parts of the original template code, hide them or add new ones. In fact, feature management in code generation is mainly used to select the desired template variant from a number of options (figure 4.13). Some examples of what can be controlled by features follow:

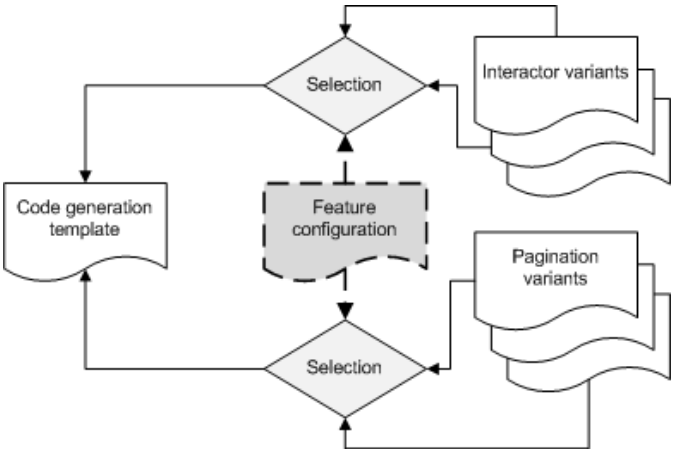

Figure 4.13: Code generator feature configuration

- Extension management (variants of application headers, toolkit/widget versions, CSS styles, etc.);
- Selection of widgets (e.g. using a slider or a number spinner for a numerical field);
- Multimedia object display (the display types of media objects can be overridden or the default display type can be customized);
- Pagination (if the there are too many interactors in one presentation, pagination or tabbing can be introduced automatically),

- Asynchronous communication types (polling or streaming);
- Event mapping (how to map the basic MARIA XML events to DOM-events);
- Semantic information types (whether to treat the value of the interactor attribute *source* as the actual content information or a reference to an external resource).

Feature management is widely used in software product line engineering. Many variant management tools exist (e.g. pure::variants[1]). These tools can be integrated in the model-driven UI development process by using plug-ins for code generation tools, for example. However, there are also very simple ways to manage variants. In some cases just a textual configuration file will suffice.

4.4.3 All-in-one templates

It is fully up to the developer of the code generation template to decide how many things should be configurable and what should be hard-coded in the template. The more options are left to feature configuration, the more universal becomes the template. Universal in terms of toolkit usage, dependencies, etc. On the other hand, it becomes complex and difficult to optimize too.

Many modern JavaScript frameworks have their own internal API, which offers performance gains compared to common generic JavaScript solutions. In some cases even the entire application structure can be changed. Using these advanced features can make the application code more compact and efficient and boost performance. On the other hand, switching to another framework will become very difficult and probably comparable to developing a new application from scratch.

With numerous different toolkits, frameworks, and widget collections available, the developer needs to decide, whether it is more important to be able to quickly switch from one to another or to use more advanced features of a selected toolkit and optimize the code generator with respect to this particular choice. Both approaches have benefits and drawbacks, so a compromise needs to be achieved in each case.

For real product line engineering flexibility is essential. In this case a universal all-in-one code generation template is, perhaps, the better choice. It will, of course, take a serious amount of time and effort to develop such a versatile template, but once it is done, variants of the output application can be created considerably faster.

If the chosen code generator engine has well-developed support for variant management, the desired flexibility may be achieved by creating variants for every single functional part of the template. In most cases, these variants will, however, need to adhere to the syntax of the template definition language, which means, that any developer willing to add a new

[1] http://www.pure-systems.com

variant will need to have at least some basic knowledge of this programming language. This can be a drawback if, for example, new widgets should be added frequently.

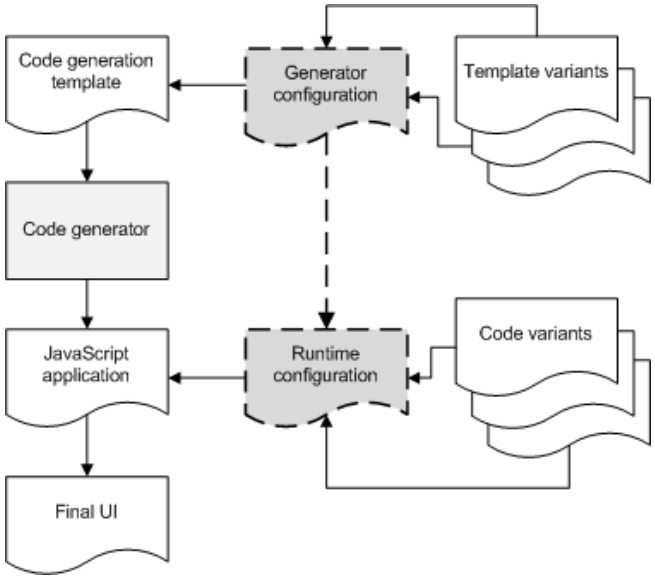

Figure 4.14: Variant management at generation time and runtime

To cope with this problem, some feature configuration can also be done at runtime in addition (or even as an alternative) to feature control at generation time. This is especially useful if the code generation template description language is not powerful enough to perform complex operations or the syntax of variant definition is too complicated. The generated application can be designed in a way that code variants are loaded dynamically by JavaScript depending on some configuration settings, which, in turn, are generated based on template variants (figure 4.14). In other words, the developer can create a JavaScript application engine, which will be configured by some data generated from the AUI.

4.5 Concept summary

This chapter introduces ways to generate AJAX-based web applications directly from slightly modified MARIA XML AUI models. The AUI elements are extended by a few new properties in order to enable custom styling and semantic content definition. The fine-tuning of the UI, normally done manually by refining the AUI to a platform specific CUI, is mainly transferred to the code generator and partly outhoused to external style sheets (like CSS). To make the generated UI easily customizable, the use of variant

management on different stages of code generation is proposed.

At the bottom line, the needed modifications of the AUI turn out to be minor (9 new attribute types and 1 new element). 6 of 9 new attributes are related to drag&drop, which is the only rich interaction type that was not supported by the AUI initially. The introduced extensions are also not directly platform specific. Although they were elaborated for the needs of AJAX-applications, they can easily be used for other platforms that support rich interactions. So the resulting modified AUI can still be seen as a platform independent generic UI description. Consequently, similar ways to generate other types of rich applications from AUIs directly can be developed.

5 Implementation

To verify the elaborated concepts a sample implementation has been developed. The primer goal was to setup a working code generator with the ability to generate applications for different JavaScript environments, based on sets of configuration files called adapters, and to generate a sample application for two different frameworks: the Dojo toolkit and pure JavaScript and XHTML as implemented in Microsoft Internet Explorer 7+.

The MDA framework openArchitectureWare was used as code generator (see section 5.2 for details). The overall scheme of the sample implementation is show in figure 5.1. The idea behind toolkit adapters is the ability to switch quickly from one target JavaScript environment to another by selecting the corresponding adapter. An adapter generally consists of two parts: the template adapter, which is only used at generation time, and the JavaScript adapter, which is needed at runtime and has to be shipped with the resulting application.

In this chapter will start with a short description with the example use case, followed by an introduction to openArchitectureWare and a brief discussion of changes to the MARIA XML AUI metamodel, which are needed in order to use it with this code generator framework (section 5.3 and 5.4). In sections 5.5 and 5.6 a closer look will be taken at the code generation templates, the non-generated JavaScript code, and the corresponding toolkit adapters. Finally, section 5.7 will compare the two included adapters: Dojo toolkit vs. plain JavaScript.

5.1 Use case: a simple cookbook

To provide a real-world test for the code generator, a simple cookbook web application will be modeled and generated. It enables the user to compose some recipes from an initial collection of ingredients. Users need to register and log in. The back-end is represented by a PHP-script. For the test all changes will only be saved for one user session for the sake of simplicity. In a real cookbook application a database would be used to store data, of course.

From the point of view of testing, the application can be divided into two important parts as follows.

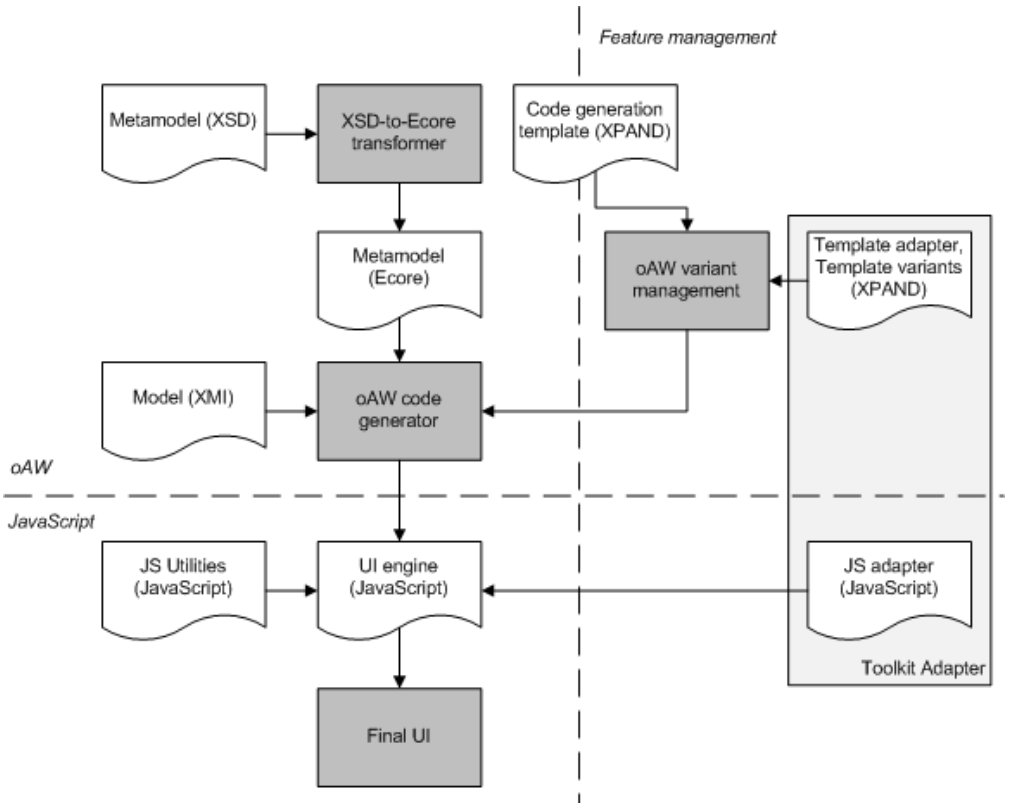

Figure 5.1: Sample implementation overview

1. The registration form (figure 5.2). Here we can test the ability of the sample implementation to generate AJAX forms with different widgets and on-the-fly input validation.
2. The cookbook application itself (figure 5.3). The interface of the application consists of a couple of blocks (like the list of available ingredients, the recipe, the recipe selector, etc.), which can be handled separately, but yet have to interact with each other. This is a good possibility to test partial UI updates and the event system.

In general, the cookbook should resemble a desktop application with instant reactions to user actions and a fluid user interface.

5.2 openArchitectureWare

OpenArchitectureWare (oAW) was chosen as code generator because it is one of the most well developed frameworks in the field, it supports XSD metamodels, it has built-in feature

Figure 5.2: Cookbook registration form screenshot

Figure 5.3: Cookbook recipe editor screenshot

management support, and a non-commercial license. oAW is a modular MDA/MDD generator framework implemented in Java. It includes a language family to check and transform models as well as generate code based on them. Supporting editors are based on the Eclipse platform. At the core there is a workflow engine allowing the definition of generator/transformation work flows. A number of prebuilt workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code.

oAW uses EMF ECore metamodels and XMI models. The XSD-based metamodel of MARIA XML AUI can be automatically transformed to an ECore-based metamodel (section 5.3), while the UI models themselves can be developed using the built-in XMI editor. Alternatively, existing MARIA XML AUI models can be easily adapted for the

use with oAW by slightly modifying the properties of the root element and translating all element names and properties from underscore to CamelCase notation.

5.3 MARIA XML as an ECore model

Although oAW can use XSD-based models like the original MARIA XML AUI, they still are internally transformed to EMF ECore models, which are then used for code generation. To simplify and quicken the work with the metamodel, this step was skipped. The XSD-based MARIA XML metamodel was transformed to an ECore-model once manually using the appropriate function of the EMF framework.

ECore models have, however, a different structure than the ones defined with XML Schema. While XSD allows to define groups of elements and to use a complex multi-level tree structure, ECore models have a linear structure, where all elements are listed. This leads to some transformation difficulties. For example, equally named elements, originally positioned in different parts of the XSD-tree cannot be directly reproduced in the linear structure of an ECore model. They need to be named differently. Accordingly, the three different types of *events* elements for different UI controls (section 4.2.5) were automatically mapped to three different elements of the ECore model (named *events*, *events1*, and *events2*) by the transformation function. Element or property groups cannot be defined either.

Neither of these issues effects the programmatic use of the model, however, they do make it less easy to read and work with for a human.

5.4 MARIA XML data model

As described in section 4.2.4, MARIA XML AUI supports XSD data type definitions. The auto generated ECore model had to be extended manually to provide support for this feature. In fact, an ECore-based model of the XSD type definition language was simply appended (figure 5.4). In this sample implementation only the most crucial elements of the language were modeled. The sample model is by no means complete. It will need to be extended if more features of the XSD type definitions are to be used.

5.5 Code generation templates

oAW has a built-in definition language for code generation templates called XPAND. It is used to define templates for every type of model element. From these definitions the code generator can produce a set of files with program code.

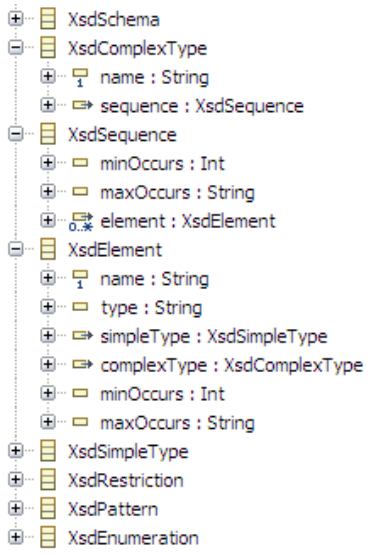

Figure 5.4: XSD type definitions modeled as ECore

Templates are enclosed in DEFINE code blocks. They expect an input parameter (model element) of a special type and can be called from other DEFINE-blocks. This makes them somewhat similar to functions in functional programming languages. In the end, an XPAND code generation template file consists of some number of these template definition blocks.

oAW supports some aspect oriented programming (AOP) techniques, which enable the developer to non-invasively modify DEFINE-blocks of existing template files (section 5.5.2).

XPAND also allows to define custom functions for model elements. These functions can be used to process the content of element attributes. As explained in sections 4.2.4 and 4.2.6, one way to handle element-to-data connections and generic script is processing appropriate references at generation time. However, XPAND's string processing functions do not support regular expressions, so quickly transforming larger strings is not easy. This has led to the decision to leave the reference strings untouched and process them later by JavaScript (section 5.6.1).

5.5.1 Template structure

To decouple the template from the JavaScript framework, the main template file was made to contain only parts of the code, which do not depend on widgets used. Concrete (X)HTML and JavaScript code for widgets was outhoused to a variant file (template

adapter) with the help of XPAND's AOP functions (figure 5.5). This type of separation enables the developer to create adapters for different JS frameworks without actually modifying the main template. Apart from this, variant files are far more compact and easy to oversee, than the template itself. Switching variant files by means of feature management makes generating an application for different frameworks fast and easy.

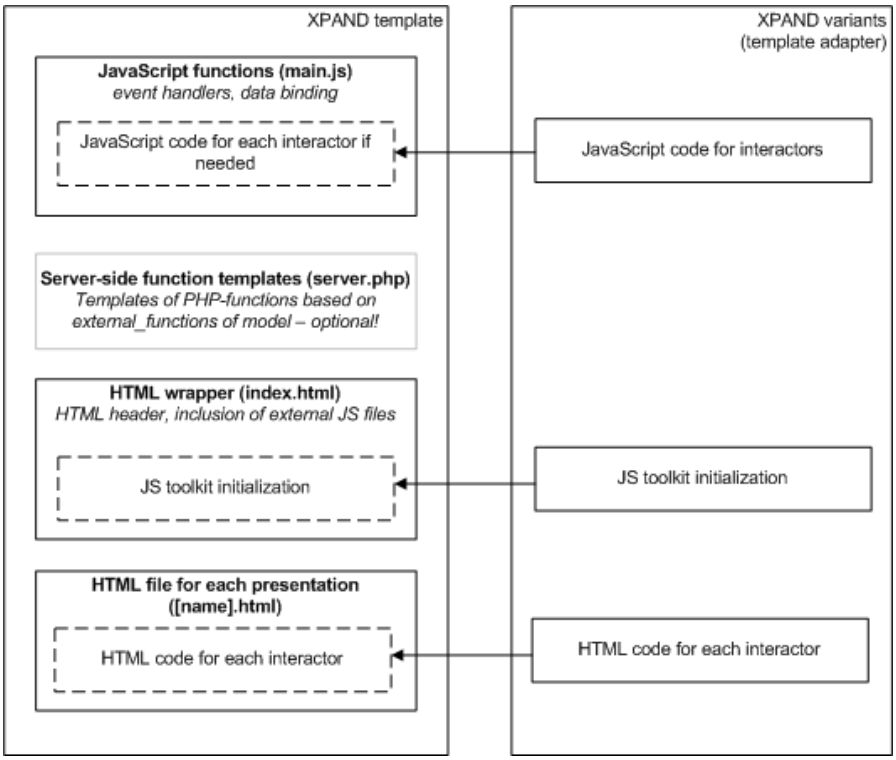

Figure 5.5: XPAND template file structure

The implemented template produces a single-page application with an external file containing all the generated JavaScript. Actually, embedded JavaScript could be used too, but in this case, each time a presentation is loaded, it would have to be checked for contained JavaScript and this script would need to be executed explicitly (by default dynamically loaded JavaScript is not executed).

Figure 5.6 shows a code excerpt from the main template file with two DEFINE-blocks for *numerical_edit_in_range* model elements. The first block takes care of the (X)HTML code. It is empty because the (X)HTML code is mostly widget-specific. The second block is responsible for generating the JavaScript code. It contains calls to other DEFINEs, which generate data references, set properties, and register events for the widget.

```
«DEFINE elementGen FOR NumericalEditInRangeType»
«ENDDEFINE»

«DEFINE elementJsGen FOR NumericalEditInRangeType»
  «EXPAND dataRefGen FOR this»«REM»take care of the dataReference«ENDREM»
  «EXPAND propertiesGen FOR this»«REM»give focus to element if needed«ENDREM»
  «REM»handle events (they're going to be separate functions«ENDREM»
  «IF this.events.eContents.size > 0-»
    «EXPAND eventGen FOR this.events.valueChange»
  «ENDIF-»
«ENDDEFINE»
```

Figure 5.6: Code generation templates for a numerical edit control in the main template file

5.5.2 Template variants

The concrete (X)HTML and JavaScript code to generate for model elements is defined by the template variants and, thus, can be controlled by variant management facilities. As mentioned before, template variants, used to generate code for a specific JavaScript toolkit, are summarized by the term "template adapter". DEFINE-blocks of the main template are replaced by respectively selected variants at generation time. In the most cases, variants will, however, contain a reference to the original DEFINE-block in addition to the variant code, so the main template's code will be executed too (see the *targetDef.proceed()* statements in the example in figure 5.7).

Very often, custom JS widgets need to be instantiated as JS variables and then have to be placed somewhere in the DOM. In most cases, they replace simpler (X)HTML elements that provide fallback functionality for browsers without JS support. Alternatively, some JavaScript toolkits (like the Dojo toolkit) introduce their own attributes for (X)HTML elements, which mark the element as an advanced widget and set its properties. In this case, the toolkit's core engine runs through the code of the page right after it finished loading and performs the required transformations upon marked elements.

Both of these ways to include JavaScript widgets in a page can be used with the proposed template structure. From each model element both JavaScript and (X)HTML code can be generated. Depending on the appropriate blocks of the variant file, the instantiation of the widget can either be done by JavaScript or by (X)HTML, or even both. If a widget can be created in both ways, the choice can also be left to the feature configuration.

Figure 5.7 shows an excerpt from the sample template adapter for the Dojo JavaScript toolkit. The first block advices (replaces) the *elementGen* template for *numerical_edit_in_range* elements, which handles (X)HTML code generation (compare to figure 5.6). The second block advices the corresponding JavaScript template (*elementJsGen*). In this particular case, the widget is created by declaratively adding some additional attributes to an

```
«AROUND template::Template::elementGen FOR NumericalEditInRangeType»
  «IF isFeatureSelected("numEdit.slider")»
    <input id="«this.id»" value="«this.value»" minimum="«this.min»" maximum="«this.max»"
      dojoType="dijit.form.HorizontalSlider" />
  «ELSE»
    <input id="«this.id»" value="«this.value»" constraints="{min:«this.min»,max:«this.max»}"
      dojoType="dijit.form.NumberSpinner" />
  «ENDIF»
  «targetDef.proceed()»
«ENDAROUND»


«AROUND template::Template2::elementJsGen FOR NumericalEditInRangeType»
  «IF isFeatureSelected("numEdit.slider")»
    dojo.require("dijit.form.HorizontalSlider");
  «ELSE»
    dojo.require("dijit.form.NumberSpinner");
  «ENDIF»
  «targetDef.proceed()»
«ENDAROUND»
```

Figure 5.7: Code variants for the implementation of a numerical edit control using the Dojo toolkit

ordinary <input> element. The JavaScript part merely loads the corresponding widget code from the toolkit's library. As mentioned before, the *targetDef.proceed()* statements tell the code generator to proceed with the original template's code shown in figure 5.6 at this place.

Note, that the variant files themselves can contain multiple variants of a single code fragment. In the above example, the *numerical_edit_in_range* element can be represented by at least two widgets from the chosen toolkit: a number spinner and a slider. Both possibilities are defined in the adapter (see the if-else statement). Which one to take, can be controlled through the configuration file: if the feature *numEdit.slider* is selected, a slider will be used, otherwise a number spinner will be generated.

OAW has a built-in textual variant management system. A list of features is stored a textual configuration file. Each option is a line of text preceded by a "+" or "-" for "on" or "off", respectively. oAW also supports more sophisticated variant management tools, like the mentioned pure::variants, on plug-in basis. To avoid possible compatibility issues, the built-in variant management was used for the sample implementation.

5.6 JavaScript

Apart from the generated JavaScript for event handlers, etc., there is also need for some support functions, which do not depend on the concrete application, but rather provide

general functionality related to MARIA XML AUI structures. Common examples are setting element attribute's values, taking care of element-to-data bindings, etc. These support functions had to be developed manually. In the sample implementation they were divided into two groups: functions, that do not explicitly depend on the chosen JavaScript toolkit and widgets, and functions, that interact with the widgets directly and, thus, provide a kind of interface adapter for them (figure 5.8).

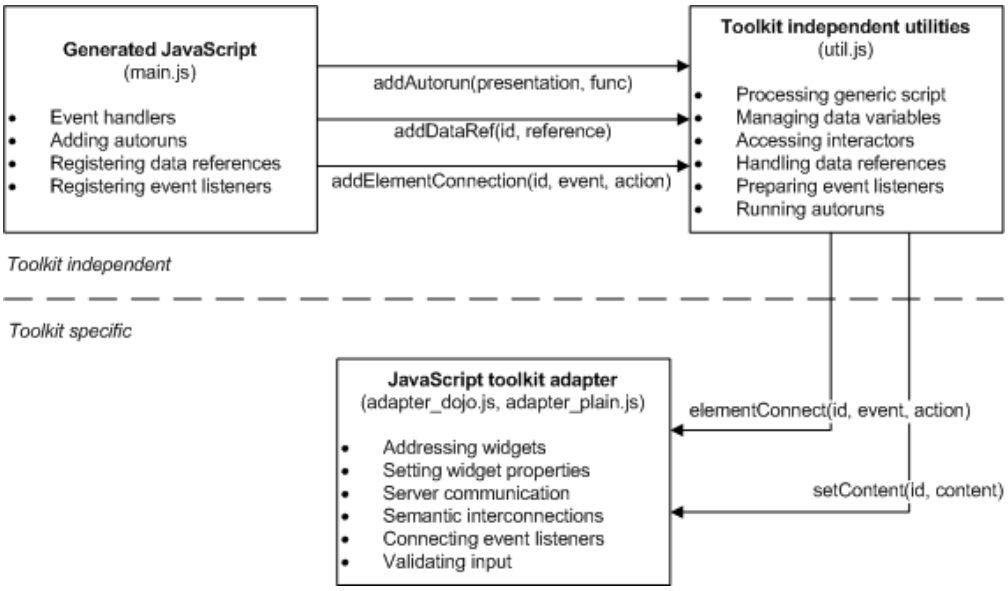

Figure 5.8: Toolkit independent and toolkit specific JavaScript

5.6.1 Toolkit independent utilities

Toolkit independent functions provide support for the MARIA XML AUI interaction model on JavaScript level. They are contained in a separate script file called *util.js*. The following main tasks are handled there.

- *Transformation of the XPath-like expressions* related to the MARIA XML AUI data model (section 4.2.4) to JavaScript variable names. In the sample implementation multidimensional arrays are used to reflect the nested structure of XPath expressions. For example, the variable name *xxx/yy/z* is transformed to the JavaScript variable *xxx['yy']['z']*.
- *Implementation of "getter" and "setter" functions* for model element attributes and data model entities, which used in the generic script of MARIA XML AUI. According to section 4.2.6, such proxy functions are used to access entities of the

UI model in order to keep the script as platform independent as possible. While reading and writing variables is not a problem, setting or retrieving properties of widgets (model elements) requires detailed knowledge of the appropriate methods supported by each widget. This is why yet another set of proxy functions has to be used, with a get- and set- function for each MARIA XML AUI element attribute like *focus*, *hidden*, *enabled*, etc. These functions are, of course, toolkit dependent and will be discussed later in section 5.6.2.

- *Loading presentations.* In general, presentations are loaded from the server using XMLHttpRequests. The concrete implementation of the request is toolkit specific since many JavaScript toolkits provide their own advanced functions for this. In the toolkit independent function the target URL is formed and the posterior actions are defined.
- *Implementation of interactor-to-data bindings.* A global register (an array with element ids and corresponding XPath variable names) is used to hold all the data references of the application's interactors. Every time an element's value is changed, the corresponding variable is updated too and vice versa. If some interactors are removed or replaced during runtime, connections to data variables are lost, but the variable values remain set for the time of the user session.
- *Connecting interactors to event handlers.* A global register (an array with element ids, event names and actions to be performed) is used to store connections between events and interactors. Each time a presentation is loaded, a toolkit specific function to register an event listener is called for every registered connection for this presentation.
- *Registering and executing actions, that need to be performed after a presentation is loaded.* There is a separate mechanism that allows registering autoruns for each presentation. These actions are performed each time the corresponding presentation is loaded. For example, if one of the presentation's interactor's focus attribute is set, it needs to be given focus after the presentation is loaded. In this case, the function call *setUi('interactor/focus', true);* needs to be added to the autorun register for the presentation. Semantic interconnections between elements (see section 4.3.3) are also handled by autorun functions. The autorun register can also be used to create event listeners. Event-to-interactor connections are handled separately just for the sake of simplicity.

Generally speaking, toolkit-independent JS functions implement the data model, generic script support, data and event bindings, and autorun functionality. These functions are an important part of the resulting application. Their concrete implementation can be changed completely without any impact upon the code generation template or the widget library used (as long as the function names, parameters, and return values remain). This

is very handy if some performance tweaking needs to be done like caching frequently loaded presentations or their data references.

In a sense, these support functions are the runtime counterpart of the main template. They are mainly used internally and called by code generated directly from the main template. Toolkit adapters will not need to make use of these functions in the most cases, so adapter developers do not even need to know much about them.

5.6.2 JavaScript toolkit adapters

JavaScript adapters provide functions to interact with the chosen JavaScript toolkit and its widgets. The most important tasks to handle are listed below.

- *Selecting a widget by its id.* Many toolkits provide their own select-by-id functions for their widgets. In some cases widgets even must be handled separately from ordinary (X)HTML elements like or <p>. If different functions need to be used for addressing widgets and other elements, some sort of separation algorithm has to be defined in the adapter. Such a problem occurred, for example, in the Dojo adapter. The current solution is a simple check to see if the element with the given id is selectable using the widget-specific function.
- *Getting and setting element properties.* Generic properties of MARIA XML AUI interactors like *focus*, *hidden*, *selected*, etc. have to be mapped to concrete properties of widgets. There is a separate function for each generic property. In some cases, it needs to distinguish between target element types. For example, the *setContent()* function will set the URL and the inner HTML in case of an <a> node, the source of an <img> node or the *innerHTML* property of a node. Additionally, some Dojo widgets have their own *setContent()* methods.
- *Handling semantic interconnections.* Semantic connections between elements can be used to create labels, descriptions, warnings, etc. How exactly they are used, is defined in the toolkit adapter. If the widgets have their own advanced labeling possibilities, these can be employed. Otherwise, perhaps, hovering descriptions can be implemented using JavaScript or the description text may simply be placed next to the element it is connected to. It is up to the toolkit adapter author to decide, how exactly semantic connections will be displayed.
- *Validating user input.* In case the interactor is connected to an element of the data model with restrictions defined (e.g. a regular expression to validate against), user input needs to be validated. How the validation is performed and how the user is notified about the result, is to be defined in the toolkit adapter. Dojo widgets, for example, have a built-in validation mechanism, so no explicit validation needs to be performed, but other widgets may not have such possibilities, so it will have to be implemented by the adapter.

- *Sending requests to the server.* This is the concrete implementation of the XML-HttpRequest for the chosen toolkit.
- *Handling multimedial content.* How embedded multimedia is displayed and what media types are supported is to be defined in the adapter. There are two functions to handle this task: one for thumbnail generation (in case the interactor's display type is set to *image*) and one for embedding. Each of them receives the id of a placeholder DOM-element and the URI to a multimedia file. Depending on the file type an appropriate multimedia container or graphical thumbnail needs to created and placed in the placeholder.

The JavaScript adapter strongly depends on the template adapter. It provides the interface for widgets, which the latter creates. This is why they can be seen as two parts of one whole (the toolkit adapter). It makes little sense to modify one without the other.

5.7 Sample adapters

The sample implementation includes two adapters: one for the Dojo JavaScript Toolkit and one for pure JavaScript and XHTML. Dojo was chosen mainly because of its rich functionality and exceptionally broad selection of tools to work with DOM elements and events. On the other hand, it still remains light-weight and practically does not depend on the server's or client's type or capabilities.

Table 5.1: Interactor implementations for Dojo and plain JavaScript+XHTML

MARIA XML interactor	Pure JS + XHTML	Dojo toolkit widgets
single_choice	<select>	dijit.form.FilteringSelect
multiple_choice	<select multiple>	dijit.form.MultiSelect
text_edit	<input>	dijit.form.ValidationTextBox
numerical_edit_full	<input>	dijit.form.NumberTextBox
numerical_edit_in_range	<input>	dijit.form.HorizontalSlider
		dijit.form.NumberSpinner
activator	<button>	dijit.form.Button
text	<p>	dijit.layout.ContentPane
grouping		dijit.layout.ContentPane
relation	<form>	<form>

Table 5.1 shows the implementation of some of the most common MARIA XML AUI interactors for the two cases in direct comparison. Dijit widgets were created by adding *dojoType* attributes to XHTML elements. However, as mentioned before, programmatic instantiation and subsequent placement in the DOM would be possible too.

Some of the main differences between the JavaScript adapters are shown in table 5.2.

Table 5.2: JavaScript adapters for Dojo and plain JavaScript

Adapter function	Pure JS + XHTML	Dojo toolkit
byId	document.getElementById()	dijit.byId or dojo.byId
elementConnect	element.eventName =	dojo.connect()
	element.setAttribute()	
setContent	element.innerHTML =	element.setContent()
send XmlHTTPRequest	browser dependent implementations	dojo.xhrGet() or dojo.xhrPost()

Using the Dojo toolkit made the JavaScript code simpler in the most cases. The implementation of the XMLHttpRequest did not need to be made browser specific, neither did the element-to-event connection function. On the other hand, some problems with the byId-functions emerged. In order to access dijit widget methods, the widgets need to be addressed by *dijit.byId()*, not *dojo.byId()*. This leads to the necessity to distinguish between the cases where widget specific functionality is needed and where the functionality of the underlying simple XHTML element suffices.

5.7.1 Semantic content and styling

Both toolkit adapters assume, that the element's content is specified in the model explicitly. For textual content the *source* attribute of the *text* element is expected to contain either plain text or HTML. For other content types, like images or multimedia, the *source* attribute of the corresponding *object* interactor is expected to contain a static URI to a local file. In the case of selection elements, the *source* attribute is expected to specify an external function, which returns a JSON array with possible values. Dynamically loaded options override the ones declaratively specified using <option> tags.

As proposed in section 4.3.1, the *style_class* attributes of interactors are expected to contain CSS class names. These are simply included in the generated XHTML code for the corresponding widgets. The style class of the root element is attached to the <body> element. In the Dojo toolkit the CSS class of the <body> element defines the default styling template to be used for all widgets. This feature resembles skinning in desktop applications.

5.7.2 Multimedia

By default, multimedial content like flash, video or audio, is displayed as a node containing a download link to the corresponding file. In order to provide the possibility

to embed multimedial content, the *embedMultimedia()* function of the JavaScript adapter is added to the autoruns for the current presentation for every *object* interactor with the *display_type* attribute set to 'media'. This function should replace the auto-generated node by the code needed to embed multimedial content for the supported media types (e.g. a JavaScript media player).

As proposed in section 4.3.1, multimedial content can either be embedded or displayed in the form of a thumbnail (or any other placeholder object, possibly, with a download link behind it), depending on the *display_type* attribute of the corresponding MARIA XML interactor. An interesting special case of the latter option is the generation of a smaller version (thumbnail) of an image. To achieve this, an interactor with an image as source and 'media' as display type can be created. The *embedMultimedia()* function of the JavaScript adapter should then generate a thumbnail from the given source. For all other types of objects, thumbnail generation is done only when the display type 'image' is selected.

5.7.3 Semantic connections

Both sample adapters move semantically connected interactors (see section 4.3.3) next to the target element. Depending on the sign of the *attach_order* attribute, they are placed before (for negative numbers) or after it (for positive numbers). If the *attach_order* attribute equals zero, the interactor's content is used as a tooltip (currently works only for attached *text* interactors).

Unfortunately, some unexpected problems occurred when trying to programmatically place Dojo widgets among other elements of the DOM. This led to strange misplacement of element with semantic connections. These problems seem to occur whenever Dijit and non-Dijit widgets are used together. In the case of pure JavaScript, programmatic element placement worked fine.

5.7.4 On-the-fly input validation

On-the-fly input validation is provided in connection with data model type restrictions based on regular expressions. Other restriction types, like special data types or enumerations of valid values, can easily be transformed to regular expressions. Every interactor, that requires validation, must be referencing a data variable with appropriate restrictions. In most cases, these restrictions are already provided by the description of expected input types of the server-side application, which is reflected in the MARIA XML AUI data model.

In the sample implementation all restrictions are stored in a general restriction register. Every time the *setData()* function is called for a variable, this register is checked and,

if applicable type restrictions are found, the adapter-specific function *validateInput()* is called.

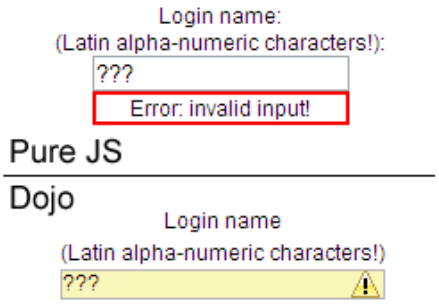

Figure 5.9: On-the-fly input validation of the sample adapters compared

The validation itself is regarded as toolkit-specific because many widgets already have built-in input validation methods. The Dojo adapter, for example, uses the built-in on-the-fly input validation API of Dojo widgets. The pure JavaScript adapter, on the other hand, simply adds some alerting text next to the form controls, which have failed validation (figure 5.9).

The same mechanism can be used to ensure correct behavior of type-specific interactors if no suitable widgets can be found. The pure JS adapter, for example, always adds appropriate restrictions to *numerical_edit_in_range* and *numerical_edit_full* interactors because the standard HTML forms do not provide numerical input controls.

5.7.5 Events

The implemented main template defines some default event mapping as shown in table 5.3. Since all three MARIA XML AUI events actually represent the activation of the corresponding interactor's main functionality, the appropriate DOM event is not selected based on the type of abstract event, but rather according to the type of the interactor.

Table 5.3: Event mapping in the sample application

MARIA XML interactor	DOM event
single_choice	onChange
multiple_choice	onChange
activator	onClick
navigator	onClick
all other types	onBlur

Neither of the sample adapters changes the default event mapping. However, it is possible to provide alternatives in the template adapter. For example, the abstract *value_change* event for *text_edit* interactors can be mapped to the *onKeyUp* DOM-event. This would cause the event to be fired after each new character typed instead of when the corresponding control looses focus.

Unfortunately, the Dojo adapter has an unresolved issue related to event registration. In some cases, the application may attempt to connect an event to an interactor more than once. This should be prevented. Otherwise, the event's action will be executed multiple times too. However, using the Dojo API, it does not seem to be possible to check for an existing connection between an event and a widget. This way it may occur that actions like loading a new presentation are performed twice in a row. For some reason, this does not happen every time. Duplicated button-labels are the most noticeable unwanted result of this issue.

The pure JavaScript adapter does not have the problem described above. However, there are compatibility issues with Gecko-based browsers (i.e. Mozilla Firefox, etc.). Concatenation of actions triggered by the same event does not work as expected. This causes the pure JS adapter to work properly only with Internet Explorer.

5.8 Sample application

Figure 5.10 shows the UI flow chart for the sample cookbook application. The login presentation is the entry point. From there the user can either enter his name and password or create a new account. After a successful login, the main application presentation is loaded, and the user can select and edit recipes.

The main presentation contains a list of available ingredients, the logout-button, and two dynamic areas, into which smaller presentations dealing with a particular actions can be loaded (figure 5.11). Area 1 normally contains some textual tip, telling the user, how to add a new ingredient to the recipe. If the user tries to do so, however, this area will show a numeric input field for the desired quantity of the ingredient. Area 2 can display a selection menu to choose the desired recipe to edit or the selected recipe ingredients.

5.8.1 Server part

The server-side PHP script implements the following functions:

getPresentation() returns the XHTML code of a presentation;

checkLogin(login,password) returns *true* if the login name and password match and *false* otherwise;

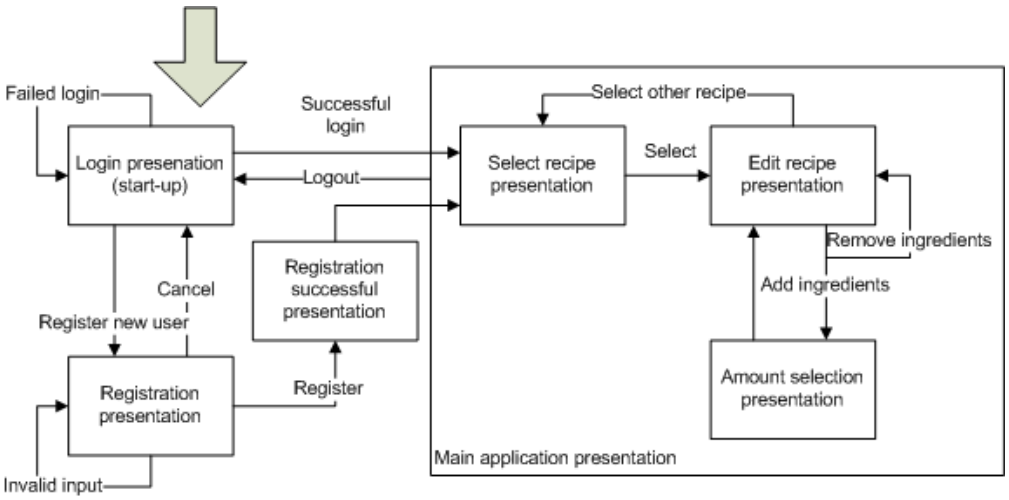

Figure 5.10: Sample application UI flow

addUser(gender,name,login,password,email) returns *true* if a new user account was successfully created and *false* otherwise;

getAvailableIngredients() returns a JSON array with ids and names of all available ingredients;

getRecipeIngredients(recipe) returns a JSON array with ingredient ids and names (names also contain quantity information) for a recipe specified by the parameter;

addIngredient(recipe,ingredient,amount) adds the specified amount of an ingredient to the selected recipe;

removeIngredient(recipe,ingredient) removes an ingredient specified by id;

getAvailableRecipes() returns a JSON array with ids and names of all available recipes.

Note, that only functions related to the inner-application logic are executed on server-side. None of these functions (except getPresentation(), of course) are related to the UI in any way.

The server-side script of the sample application is based on a template, which was generated from the model's external function specifications. The generated template contains function stubs with some simple PhpDoc.

5.8.2 UI structure

The generated UI consists of an index file and separate XHTML files for every presentation. The XHTML only lists the interactors (figure 5.12 shows the code for the main application

Figure 5.11: Functional areas of the cookbook application

presentation). All the dynamic widget properties, event connections, dynamic content, etc. is added by JavaScript later. Also note the <p> elements with text describing the controls. These will be transformed to labels later on. Initially, they can be placed anywhere in the presentation.

Generated JavaScript is stored in the file main.js. It contains

- vent handler functions;
- functions, loading presentations according to MARIA XML connections;
- calls to the *addDataRef()* function, registering interactor-to-data references;
- calls to the *addElementConnection()* function, registering event listeners;
- calls to the *addValidation()* function, registering data type restrictions for variables;
- calls to the *addAutorun()* function, registering code to run upon presentation load;
- calls to the *embedMultimedia()* and *showImage()* functions, dealing with multimedial content and images.

Once this file is loaded, autoruns for the initial presentation are executed. Autoruns play a key role in applications produced by the sample code generator. The reason why most of the widget properties must be set by JavaScript is the wish to support declarative as well as programmatic instantiation of widgets. Especially in the latter case, all widget-specific properties can only be set after the widget itself has been properly deployed.

```
<?xml version="1.0" encoding="windows-1251"?>
<div id="mainAppPresentation">
  <div id="mainAppFrame">
    <div id="ingredientListFrame">
      <div id="ingredientListHelpContainer"> </div>
      <div id="ingredientListContainer">
        <select id="ingredientList" multiple="multiple">
          <option value="product1">Product 1</option>
          <option value="product2">Product 2</option>
          <option value="product3">Product 3</option>
        </select>
      </div>
      <p id="ingredientAddToRecipeButtonLabel" class="">
        Add selected to recipe
      </p>
      <button id="ingredientAddToRecipeButton"> </button>
    </div>
    <div id="recipeFrame"> </div>
    <div id="logoutButtonFrame"> </div>
  </div>
</div>
```

Figure 5.12: XHTML code for the cookbook's main presentation

5.8.3 Nested presentations and partial UI updates

Asynchronous partial UI updates are, perhaps, the most important feature of AJAX. They make the UI seamless and desktop-like. They also enable the designer to split the UI into smaller parts and reuse them. The main presentation of the cookbook application is a good example for a composed UI. The MARIA XML *grouping* element "ingredientListHelpContainer" represented by corresponding element will be filled with the simple text tip after the presentation is loaded. If the add-ingredient-button is pressed, this element will contain the quantity input field with the corresponding buttons (figure 5.11). Each of the filling options is a separate presentation. After the desired ingredient is added to the recipe, the element's contents will be reverted to the initial state, and the contents of the recipe ingredient listing will be updated.

As explained in section 4.2.2, MARIA XML conditional connections can be used to describe this type of behavior. Connections used in combination with event handlers attached to the same interactor provide the possibility to perform some actions and a UI

update in one step. Figure 5.13 shows such a combined action for the above example. The numerical labels represent the execution order.

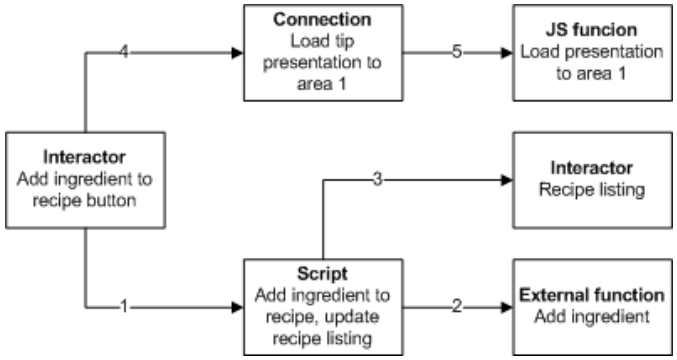

Figure 5.13: Combined action with partial UI update

Connection actions are always executed after the event-bound script of the interactor because conditional connections may depend on data variables affected by this script. Figure 5.14 shows the control flow chart of the login button's actions to illustrate the problem.

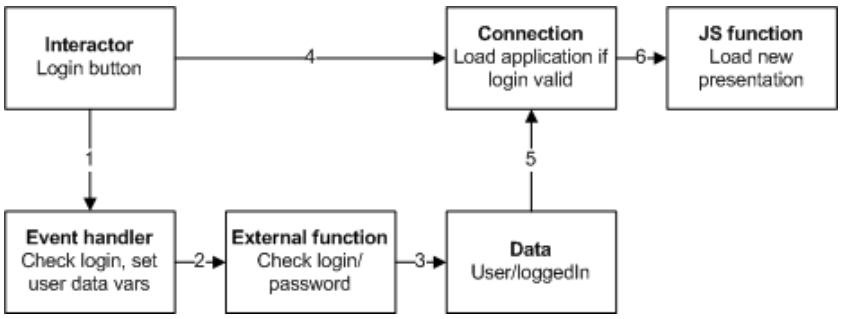

Figure 5.14: Data dependent conditional connection

Additionally, external function calls cannot be asynchronous because the generator cannot find out, if other parts of the script depend on their output. In the above example, an asynchronous password check would mean, that with great probability the *user/loggedIn* variable would only be set after the connection performs its condition check. Loading the new presentation, on the other hand, can be done asynchronously if the immediate post-processing includes the autorun functions.

5.8.4 Validating the registration form

The registration presentation contains a simple form, which needs to be submitted in order to register a new user. The server script expects the login and password fields to accept Latin alpha-numeric characters and the email field to hold a valid email address. These requirements are reflected in regular expressions set for corresponding elements of the sample application's UI data model. Should the user input invalid data, he will be notified immediately. The "Register" button can still be pressed, but its event handler will receive a negative reply from the server when attempting to register the user and will not set the conditions to proceed to the next presentation (figure 5.14).

5.8.5 Application variants

Figure 5.15: Pure JavaScript and Dojo implementations of the cookbook in comparison

At the moment, three variants of the example cookbook application can be generated:

- A pure JavaScript version;
- A Dojo version with numerical input controls represented by sliders;
- A Dojo version with numerical input controls displayed as number spinners;

Accordingly, there are three options to set in the configuration file (config.txt): *useAdapterPlain*, *useAdapterDojo*, and *numEdit.slider*. In this way, the choice of the adapter is controlled by a single configuration value. Figure 5.15 shows the login screen of the two variants in direct comparison. Note, that the built-in feature manager of oAW does not support exclusive choices, so there is now way to ensure, that only one of the two adapter-options is set. More advanced variant management tools support relations between options.

5.9 Implementation summary

The sample implementation is capable of generating AJAX applications based on pure JavaScript and XHTML or the Dojo JavaScript toolkit from MARIA XML AUI models. For each target platform a special toolkit adapter is used. Switching adapters can be done by means of oAW feature management. Each adapter consists of a template variant file (template adapter) and a JavaScript interface for the chosen toolkit (JavaScript adapter). New adapters for other AJAX frameworks can be added easily.

The JavaScript implementation of MARIA XML AUI model constructs is outhoused to a special JS library. It can be changed independently. This offers additional possibilities in terms of performance optimization, security, etc. It is clearly separated from the adapters. Although its functions can be used by an adapter too, there is normally no need to do so. Consequently, potential adapter developers will not inevitably need to have extensive knowledge of the internal structure of the code generation templates.

6 Evaluation

After having validated the proposed concept of direct AJAX UI code generation from an abstract UI description, it is now possible to compare this approach to manual coding and the traditional model-driven UI development involving a second concrete UI model.

This chapter will summarize and evaluate the results of this work. After a general overview in section 6.1, the proposed technique will be compared to the AUI-CUI-FUI approach and to manual development in sections 6.2 and 6.3, respectively. Section 6.4 will follow with a discussion of the main problems encountered. Finally, section 6.5 will present a brief overview of possible future work.

6.1 Primary results overview

The primer goal of this work is to evaluate the possibility of generating a highly-responsive AJAX UI from an abstract UI description model based on the MARIA XML AUI language. In section 1.2 the requirements for such a UI are defined as follows.

- *XHTML + JavaScript as client-side technology.* The generation of XHTML markup and JavaScript code from MARIA XML AUI constructs is discussed in detail in section 4.2. In 4.3 extensions to the AUI model are proposed in order to provide definition possibilities for styling, semantics, and rich interactions.
- *Immediate response to user actions (where desired).* High responsiveness of the UI is provided by the combination of event-based actions, a generic scripting language, and behind-the-scenes communication with the server (sections 4.2.5, 4.2.6, and 4.2.7). The use of advanced widgets like numerical controls, filtering selects, etc. can enhance the UI flow even further. On-the-fly input validation adds a great deal of interactivity to such a common task as form filling. How it can be achieved using the MARIA XML data model (section 4.2.4), is explained in sections 5.7.4 and 5.8.4.
- *Partial content refreshes.* As discussed in sections 4.2.1 and 4.2.2, partial content refreshes can be modeled by MARIA XML AUI conditional connections. They allow one to load a new presentation into any composition interactor (*grouping* or *relation*) of another one. A concrete implementation is presented in 5.8.3 in detail.

- *Asynchronous communication (streaming).* The concepts of continuous updates of selected UI elements are introduced in section 4.2.8. The concrete implementation largely depends on the requirements for the application to be developed and the chosen target environment (JavaScript toolkit).
- *Customizable appearance (CSS).* Visual styling and look&feel are completely outhoused to CSS style sheets. To link model elements to the correct styles, additional interactor attributes are introduced in section 4.3.1.
- *Rich interactions (e.g. drag&drop).* Drag&drop was found to be the only type of rich interactions that could not be modeled by MARIA XML AUI initially (although, the definition of rich interactions is very inexact). To close this gap, a special extension to the AUI is proposed in section 4.3.4.

With the CUI model being skipped, the code generator template's role increases. It now defines, how exactly abstract AUI interactors are transformed to concrete widgets. Section 4.4 proposes feature management as a way to customize code generation. In connection with a special JavaScript runtime engine (section 4.4.3) some very flexible code generation environments can be set up, capable of producing different final UIs depending on simple configuration files.

Chapter 5 gives a detailed overview of an implementation of such a universal AJAX code generation environment based on openArchitectureWare as code generator and a custom JavaScript runtime engine. Using combined template and JavaScript adapters (sections 5.5 and 5.6, respectively), it is possible to generate UI code for different JavaScript toolkits and widget libraries simply by changing configuration options. The adapters themselves can contain multiple variants of widget code, which can also be selected by feature configuration.

To demonstrate the power of a configurable code generator, the sample implementation includes adapters for pure JavaScript and XHTML and the Dojo JavaScript toolkit (section 5.7). Depending on the adapter used, the sample cookbook application (section 5.1) is either composed of Dojo widgets or of ordinary HTML form controls. Section 5.8 describes this in detail.

6.2 Configurable code generator vs. CUI

Figure 6.1 shows a comparison between the traditional AUI-CUI-approach and direct code generation for multi-platform code generation environments. Instead of a code generation template, producing code for a specific framework from the CUI, the toolkit adapter was introduced, which extends the largely framework-independent code generation template and provides a runtime engine. The styling and spatial arrangement information is held in separate style sheets.

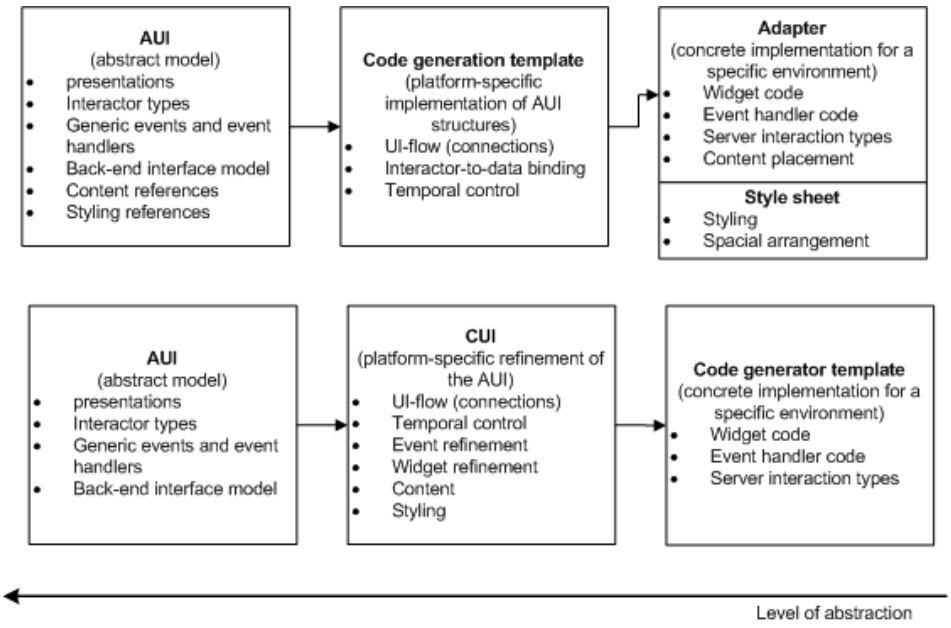

Figure 6.1: CUI vs. configurable code generator with adapters

Both approaches may seem similar. The main difference, however, is that the CUI is yet another model. It is application specific and needs to be separately created for every AUI and for every target environment (that would make two different CUI models if the application is to be implemented for two JavaScript frameworks). Moreover, all CUIs must be changed if significant changes to the AUI or the underlying platform are introduced.

The direct code generation approach does not have these problems. The AUI is the only model, so there are no problems with consistency. The code generation template only defines general application structure for the target platform (in this particular case, JavaScript and XHTML) and all the toolkit adaptation is done by the respective adapters. The internal logic of the template and the adapters can be changed totally independently.

Finally, the CUI mostly requires human attention even if the code generator is well set up. In the case of direct code generation, well-developed templates and adapters are enough to automatically generate FUI code from arbitrary AUIs. Configuration of the code generator still needs to be done manually, but it does not directly depend on the AUI used.

On the other hand, an additional platform-specific UI model leaves more space for fine-tuning and customizing the UI. For example, one could set a specific numeric interactor to be a slider and its neighbor to be displayed as a simple input field. This is generally

not possible when generating code from the AUI. Even with appropriate configuration options one would only be able to change the look&feel of all interactors of a certain type at once. This particular problem is partly solved by the introduction of additional styling attributes (section 4.3.1) to AUI interactors, but there are other fine tweaks like event mapping, table-like arrangement, etc. that remain.

Actually, the AUI itself mostly requires some manual refinement in both cases. Although much of it can be derived from the specifications of server-side applications (section 3.2), forming presentations, setting conditions for connections, etc. cannot be automated in most cases. If direct code generation is used, custom styling and possible manual content adaptation needs to be done in the AUI additionally. Nevertheless, an extended AUI leads to less human efforts than two separate UI models, expressed with different vocabulary and edited by different authoring tools.

6.3 Efficiency and performance of generated UIs

In order to estimate the code efficiency and performance of the generated UIs, they were compared to an identically looking cookbook example UI for the same server-side script, which was written manually. This reference implementation uses Dojo widgets and parts of the Dojo API.

Table 6.1 shows the sums of HTML and JavaScript code lines and the total application sizes for all three variants of the cookbook application. In order to provide a more reliable result, JavaScript files were minified[1] before their size was determined. Lines of code, on the other hand, were counted in non-minified files.

Table 6.1: Sample application comparison for generated and manually developed UIs

	Generated pure JS	Generated Dojo	Manual Dojo
HTML lines of code	159	161	114
JS lines of code	627	651	212
Total lines of code	786	812	326
HTML size	6,7 KB	9,2 KB	6,6 KB
JS size (minified)	18,9 KB	19,4 KB	5,6 KB
Total size (minified)	25,6 KB	28,6 KB	12,2 KB

This comparison clearly shows the redundancy of generated code. The hand written UI is less than half the size and also only has half as many lines of code. The difference in HTML code can be explained by the use of *text* interactors as labels for other interactor

[1] http://www.vlead.in/resources/tools/minify-javascript/index.php

types in the AUI. In the hand written code, labels were explicitly declared in the markup. The greatest difference, however, is in the JavaScript part. The generated UI has almost four times as much script as the reference implementation. This huge difference is a tribute to universality. The generated code is structured in such a way, that it can theoretically be used with any toolkit with a suitable adapter. The hand written UI, on the contrary, is Dojo specific and uses shortcut functions provided by the Dojo API to compact the code. The code can also be organized in a more clever way if the developer knows the true purpose of every UI element. The code generator cannot know it, so it has to treat all elements equally.

In order to compare performance, the time needed to load the application was measured. Table 6.3 shows test results for generated and manually developed Dojo-based UIs. The tests were performed on a Windows XP SP3 notebook based on the Intel CentrinoPRO platform with 2,4 GHz (T7700) and 2 GB of RAM. Both, client (Microsoft Internet Explorer 8) and server (Apache 2.2.0) were located on the same system. The time measurement started on the first line of the HTML header and ended with the *onLoad*-event fired at the <body> element. For each measurement the page was refreshed, thus, excluding the possibility of caching. For every test 30 measurements were performed. Table 6.3 displays the respective average values and standard deviations. Non-minified JavaScript files were used.

Table 6.2: Load time comparison for generated and manually developed UIs in ms

	single_choice	text_edit	activator	standalone text	connected text	object	grouping
%	5	25	10	10	35	5	10

Synthetically composed presentations based on multiple registration forms from the cookbook application were used for tests with large numbers of interactors. Interactors used in these presentations and their relative quantity (in percent of the total amount of interactors) are listed in table 6.2.

Table 6.3: Loading time comparison for generated and manually developed UIs

	Generated Dojo			Manual Dojo		
	Mean (ms)	StDev (ms)	JS size (KB)	Mean (ms)	StDev (ms)	JS size (KB)
cookbook application (9 presentations, 73 interactors total)	376	79	33,6	206	63	5,6
1 presentation with 100 interactors	1405	55,9	39,8	180	52,5	9,8
1 presentation with 200 interactors	2580	40,3	59,5	201	44,5	12,5
1 presentation with 300 interactors	3942	114,7	79,3	215	33,8	14,9
3 presentations, 100 interactors each	1467	54,7	76,4	186	44,8	16,2

The results of the test show, that the presentation load times of generated UIs largely depend on the quantity of JavaScript to be processed. Since all element properties and especially semantic connections in the generated application are handled by JavaScript, performance almost linearly depends on the quantity of interactors per presentation. Note, that an application with 3 presentations of 100 interactors each, takes about the same time to load as an application with only one presentation of this size. A single presentation with 300 interactors, on the contrary, needs almost three times as long.

The manually developed version of the UI does not show any noticeable performance drop with rising UI element numbers. Most of the element's properties are set declaratively in the HTML code, so the amount of JavaScript to be processed does not increase much either. Additionally, JavaScript event handlers and other functions can be reused instead of being replicated for every interactor.

It should also be pointed out, that with 300 interactors the total size of the generated JavaScript files exceeded 70KB. If this file had to be downloaded from a remote server, this would, perhaps, be another huge performance decreasing factor.

All in all, the presented test results show that the generated UI code has a very high redundancy. Especially the JavaScript code. With the current version of the code generator, generated UIs are at least twice as big as comparable hand-written ones. However, for simple applications with a moderate number of interactors per presentation, the difference in size does not affect performance in a critical manner. Reaction times of about 300 ms are quite acceptable for most purposes. One should also keep in mind, that the initial UI load time, which was measured in the presented tests, includes the time needed to download the main JavaScript file. Once it has been cached by the browser, response times will shorten.

6.4 Main limitations

Throughout this work, and especially during the development of the sample implementation, some limitations of the approach came into light.

First of all, the creation of a universal code generation environment, which would suite as many different application models as possible, demands huge efforts. The development of code generation templates and JavaScript adapters takes much more time, than manual development of a comparatively simple AJAX UI. On the other hand, once the template is good enough, the development of new UIs is indeed much quicker.

The generated UIs, however, still have some limitations. For example, MARIA XML AUI lacks proper temporal control (the use of additional CTT models could resolve this problem, but this would introduce yet another model to be created). In other words, there is no way to define, which action comes first, which second, and which can be run in parallel. For dynamic AJAX UIs with asynchronous requests to the server, this is quite important. In fact, server interaction wrapped in external function calls cannot be asynchronous at all, because very frequently subsequent actions of the UI depend on its results (see section 5.8.3 for a detailed example). This is not really a problem for simple server requests. However, if the execution of the server-side function will take significant time, the UI will simply block for this period, which is something AJAX techniques actually aim to avoid.

Another concern is the generic scripting language in the AUI. It enables the developer to define very versatile event handlers (which are essential for rich interfaces), but it produces significant overhead too. Especially if it is interpreted at runtime, as in the case of the sample implementation. In this work the syntax of the generic scripting language was assumed to be equal to JavaScript (section 4.2.6). This simplified the execution of generic script, but made the model if not platform-dependent then, at least, optimized for JavaScript platforms. Once again, this problem concerns mainly larger applications with numerous and possibly complex script actions. On the other hand, generic script also introduces another limitation: while it is possible to create a visual tool, which would enable a non-programmer to compose the interface without special knowledge of any programming language, script-based actions would require at least the knowledge of the scripting language itself.

Perhaps, part of the problem is the pre-release version of MARIA XML, which used in this work. In future versions some of these issues may be solved. Especially since the latest MARIA XML related publication [PSS09] speaks about temporal operators, a dialog model, and other features, that could not be discovered in the present version 1.3. A detailed documentation and a visual authoring tool are also things still to do.

In general, the lack of proper authoring tools is one of the most serious drawbacks. At present, there is still no MARIA XML visual editor openly available. At the same time, a

visual aid would save a huge amount of time when developing new UI models. Even such simple UI models, as the one for the cookbook example, very quickly become difficult to oversee. One has to keep in mind all id's of related interactors, etc. OpenArchitectureWare editors used in the sample implementation are a good help, but they have a lot of problems of their own. Although some video tutorials exist, there is no in-depth documentation of the model or the template editors. The latter also permanently show non-existent errors in the template code.

The XPAND template definition language is very poorly documented and difficult to use too. In fact, it turned out to be much easier to outhouse certain functions to the JavaScript adapter, than to try to implement them correctly in XPAND. XPAND is normally just used to register commands, which are later executed by JavaScript at runtime. This method is at least partly responsible for the size of the generated files. Additionally, generated code has very many empty lines. Almost 15% (3KB) of the cookbook's generated JavaScript are empty space. In the case of XHTML, the included XML-beautifier takes care of similar problems, but there is no JavaScript beautifier available, and the corresponding Java tool does not seem to work with JavaScript.

To summarize the above, a list of the main problems encountered follows:

- Undocumented pre-release version of MARIA XML;
- No specialized MARIA XML authoring tool;
- Poor documentation for oAW and XPAND;
- XPAND not powerful enough to perform complex string transformations;
- Poor quality of oAW editors.

At the bottom line, it should be pointed out, that well-developed editors and authoring tools are truly essential for an efficient model-driven approach to UI development. Since rapid prototyping and product line engineering are some of the main fields of use for abstract UI descriptions, development time is an important factor. And time can be saved just by providing better editor environments.

6.5 Outlook

The sample implementation described in chapter 5 is mainly a proof of concept. As mentioned before, it relies on a pre-release version of MARIA XML (version 1.3). Once the final version is released, the code generation environment can be updated with more detailed implementations of all the features of MARIA XML AUI. Performance and code efficiency can be improved by introducing additional optimizations like caching or post-processing. Most of all, however, a decent authoring tool must be developed. In

connection with such a tool and a collection of adapters for the most common JavaScript toolkits (like Yahoo UI, for example), this code generator can successfully be used for real rapid prototyping or fast development of simple AJAX applications.

To avoid the problems connected with oAW, an alternative implementation based on another code generator can be developed too. An interesting option may be the use of XSLT. Current versions of this language also support string processing functions with regular expressions. Perhaps, it will be possible to transform MARIA XML generic script and data references to corresponding JavaScript code at generation time. This will surely have a positive impact on the size of generated code.

In the long run, model-driven UI generation will surely take its place in the development of the next generation of web applications. Whether based on one or two models, it needs some improvement and foremost good support tools to be developed. It is very probable, that both approaches will evolve and simply serve different purposes. From this point of view, it would undoubtedly be very interesting to see a direct comparison of the two code generation techniques.

7 Conclusions

With the concepts proposed in this thesis it is possible to generate highly interactive AJAX UIs from platform-independent abstract UI description models of MARIA XML with minor extensions. In contrast to the traditional 2-step approach, involving abstract and concrete UI models, direct code generation from the AUI ensures consistency at all times. It also requires less human interference. Moreover, the code generation process can be entirely controlled by feature management techniques. This allows one to switch easily between different widget libraries or even entire JavaScript toolkits merely by changing a few configuration options.

As proof of concept a sample implementation of a complete code generation environment based on openArchitectureWare is presented. The sample implementation has a modular structure, which allows one to extend its support for JavaScript libraries in future. For the time being, it includes adapters for the generation of two variants of AJAX UIs: pure JavaScript + XHTML and UIs utilizing the Dojo JavaScript toolkit.

The generated UIs fulfill the requirements for AJAX applications introduced in the beginning of this work. However, the resulting code shows a high level of redundancy compared to analogous manually written UIs. This is the price paid for a universal code generator for platform-independent models. In general, the more generic the model is, the less the generated application can take advantage of special features of the underlying platform, the less efficient it is. This is not a problem of MARIA XML or the presented approach specifically, but rather a general shortcoming of all abstract descriptions.

However, code efficiency is by no means the aim of model-driven software development. The main advantage of code generation is the speed, with which new applications can be implemented, and the ease of variation.

With a well-developed code generator and decent authoring tools available, the proposed approach will surely be advantageous for the following two major tasks.

- *Quick development of small and middle-sized applications* with moderate demands to performance and code efficiency. Common examples are electronic forms, simple booking systems, and other applications that require fast and convenient data input. Especially service-oriented applications can benefit, since the AUIs can be largely automatically derived from web service descriptions of different kinds.

- *Rapid prototyping and product line engineering.* With a code generation environment similar to the presented sample implementation, it is possible to generate different UI variants from a single model very quickly. After a suitable variant is selected, it can be manually optimized and enhanced if needed.

For bigger applications with a branched UI flow, complex interconnections between elements or high performance demands, the traditional AUI-CUI-FUI approach is, perhaps, the better choice for model-driven development. The intermediate CUI leaves more space for manual fine-tuning before code generation.

For a single complex AJAX application to be developed and no ready code generation environment available, manual coding still remains the quickest and easiest option for UI development. Model-driven techniques are only advantageous if at least multiple variants of the application's UI are required.

Perhaps, if a working modular AJAX code generation environment, like the one presented in this work, and versatile adapters for different toolkits and code libraries, and a good set of authoring tools will become widely available in future, model-driven UI development will come to be more popular, efficient and accessible.

A Appendix

A.1 Content of enclosed CD

The enclosed CD contains an electronic (PDF) version of this thesis and source codes of code generation templates and adapters. The content of the root folder is listed below.

- Distrib. This folder contains the Windows distributions of oAW and XAMPP (an Apache-based web server).
- oAW Project. The sample openArchitectureWare project.
- Source code. This folder contains the source code of the two sample toolkit adapters, the extended MARIA XML AUI metamodel (XSD), the main code generation template, and the corresponding JavaScript utilities.
- Testbench. Copy these files along with any generated code to your web server to test the generated UI.
- Tests. A collection of UIs, used for testing purposes throughout this work.

A.2 Setting up the code generator

To set up the sample code generation environment, please perform the following steps.

1. Download and install openArchitectureWare (http://www.openarchitectureware.org). A Windows distribution can also be found in the *Distrib* folder on the enclosed CD.
2. Copy the folder oAW Project to your hard drive.
3. Start oAW and select the oAW Project folder as your workspace.
4. In the now opened workspace, enter the *firstTest* project, select *src/workflow*, and run *generator.oaw* (figure A.1).

You can manage available features by modifying the file config.txt.

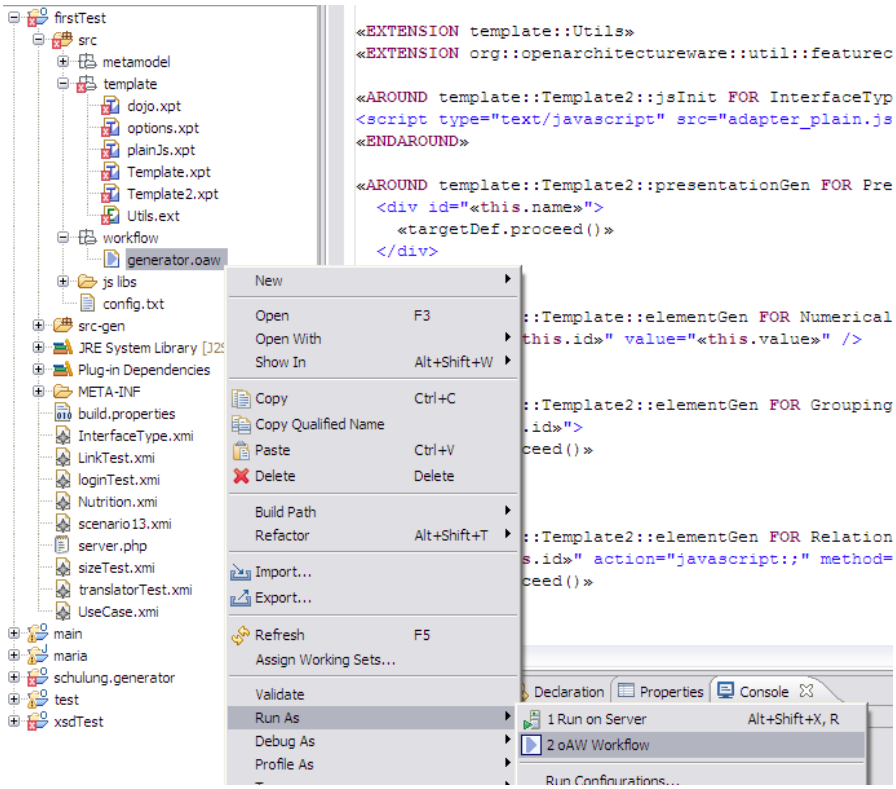

Figure A.1: Run code generation oAW screenshot

A.3 Testing the generated UI

The generated UI itself can be run on InternetExplorer 7+. Mozilla is currently not supported. However, in order to run the sample cookbook application, a web server is needed. During this work tests were performed on the XAMPP distribution of the Apache Server. A copy of this distribution for Windows can be found in the *Distrib* folder of the enclosed CD. To run the cookbook application, follow the steps below.

1. Start a web server. If using the enclosed XAMPP distribution, unpack the corresponding archive to *C:\xampp* and run *xampp_control.exe*.
2. Copy the Testbench folder from the CD to the public HTML folder of your server (*C:\xampp\htdocs* for the enclosed distribution).
3. Copy the generated UI (excluding the *server.php* stub) from the src-gen subfolder of the *firstTest* oAW project to the same folder.
4. Open the web application by directing your browser to *http://localhost/Testbench/*.

Bibliography

[BCFC06] A. Bozzon, S. Comai, P. Fraternali, and G. Toffetti Carughi. Conceptual modeling and code generation for rich internet applications. *International Conference on Web Engineering*, pages 353–360, 2006.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33:137–157, 2000.

[Con03] Jim Conallen. *Building Web Applications with UML (2nd Edition)*. Addison-Wesley, 2003.

[FZ98] M. Franklin and S. Zdonik. Data in your face: Push technology in perspective. In *Proc. of ACM SIGMOD International Conference on Management of Data SIGMOD 98*, 1998.

[Gar05] Jesse Garrett. Ajax: A new approach to web applications., 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[KK02] N. Koch and A. Kraus. The expressive power of uml-based web engineering. In *IWWOST 02: 2nd International Workshop on Web-oriented Software Technology*, 2002.

[KK03] N. Koch and A. Kraus. Towards a common metamodel for the development of web applications. In *ICWE 03: International Conference on Web Engineering*, 2003.

[KK06] N. Koch and A. Kraus. Transformation techniques in the model-driven development process of uwe. In *MDWE 06: 2nd Model-Driven Web Engineering Workshop*, 2006.

[LLPS08] Marco Lerro, Roberto Longobardi, Gianluca Perreca, and Alessandro Scotti. The abstract user interface markup language web toolkit: An auiml renderer for javascript and dojo. *IBM developerWorks*, 2008.

[LPSF07] M. Linaje, J.C. Preciado, and F. Sanchez-Figueroa. A method for model based design of rich internet application interactive user interfaces. In *International Conference on Web Engineering*, 2007.

[MAMRVO06] J. Muñoz Arteaga, Francisco J. Martínez-Ruiz, J. Vanderdonckt, and A. Ochoa. Categorization of rich internet applications based on similitude criteria. In *XI Simpósio de Informática e VI Mostra de Software Acadêmico da PUCRS*, 2006.

[MM03] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0.1, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[MR07] Francisco Javier Martínez-Ruiz. A development method for user interfaces of rich internet applications. Master's thesis, Universite catholique de Louvain, 2007.

[MRVA08] Francisco J. Martínez-Ruiz, Jean Vanderdonckt, and Jaime Muñoz Arteaga. Web user interface generation for multiple platforms. In *ICWE 2008 Workshops, 7th Int. Workshop on Web-Oriented Software Technologies. IWWOST 2008*, 2008.

[OMG07] OMG. Mof 2.0 / xmi mapping specification, v2.1.1, 2007. http://www.omg.org/technology/documents/formal/xmi.htm.

[Pat99] F. Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, 1999.

[PLCSF07] J.C. Preciado, M. Linaje, S. Comai, and F. Sanchez-Figueroa. Designing rich internet applications with web engineering methodologies. *Proceedings of the 9th IEEE International Symposium on Web Site Evolution*, pages 23–30, 2007.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction level language for service-oriented applications in ubiquitous environments. *TOCHI special issue on User Interface Description Languages for Next Generation User Interfaces*, 2009.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[TC07] G. Toffetti Carughi. Modeling data-intensive rich internet applications with server push support. *Int. workshop Model-Driven Web Engineering in conjunction with ICWE*, 2007.

[VG08] S.V. Vahid Gharavi. Model-driven development of ajax web applications. Master's thesis, Delft University of Technology, 2008.

[W3C99] W3C. Xml path language (xpath), 1999. http://www.w3.org/TR/xpath.

[W3C01] W3C. Xml schema, 2001. http://www.w3.org/XML/Schema.

[W3C05a] W3C. Document object model (dom), 2005. http://www.w3.org/DOM/.

[W3C05b] W3C. Web service semantics - wsdl-s, 2005. http://www.w3.org/Submission/WSDL-S/.

[W3C06] W3C. The xmlhttprequest object, 2006. http://www.w3.org/TR/XMLHttpRequest/.

[W3C07] W3C. Xsl transformations (xslt) version 2.0, 01 2007. http://www.w3.org/TR/xslt20/.

[Wik09] Wikipedia. Meta-object facility, 08 2009. http://en.wikipedia.org/wiki/Meta-Object_Facility.

[ZK] Zk architecture overview. http://docs.zkoss.org/wiki/ZK_Architecture_Overview.

List of Figures

2.1 Example of PSMs and code being generated for different platforms from a single PIM [VG08] . . . 6
2.2 Illustration of the Meta-Object Facility [Wik09] . . . 8
2.3 Asynchronous communication in AJAX [Gar05] . . . 13

3.1 RUX-model (http://ruxproject.org) . . . 15
3.2 ZK framework architecture [ZK] . . . 17
3.3 AUIML Web Tools architecture [LLPS08] . . . 18
3.4 General design-flow for model-driven web application development . . . 20
3.5 Design-flow for web application UI with direct code generation from AUI . . . 22
3.6 MARIA XML AUI overview [PSS09] . . . 24
3.7 MARIA XML AUI code example . . . 25

4.1 UI code generation overview . . . 28
4.2 MARIA XML presentation element . . . 29
4.3 MARIA XML connections . . . 31
4.4 MARIA XML interactors overview [PSS09] . . . 32
4.5 MARIA XML *numerical_edit_in_range* interactor . . . 35
4.6 MARIA XML event handlers . . . 37
4.7 MARIA XML AUI event handler code example . . . 38
4.8 MARIA XML generic script example . . . 39
4.9 MARIA XML external function example . . . 41
4.10 Extended *text* interactor . . . 44
4.11 Extended *single_choice* interactor . . . 46
4.12 Code generation template structure . . . 48
4.13 Code generator feature configuration . . . 49
4.14 Variant management at generation time and runtime . . . 51

5.1 Sample implementation overview . . . 54
5.2 Cookbook registration form screenshot . . . 55
5.3 Cookbook recipe editor screenshot . . . 55
5.4 XSD type definitions modeled as ECore . . . 57
5.5 XPAND template file structure . . . 58

5.6 Code generation templates for a numerical edit control in the main template file . . . 59
5.7 Code variants for the implementation of a numerical edit control using the Dojo toolkit . . . 60
5.8 Toolkit independent and toolkit specific JavaScript . . . 61
5.9 On-the-fly input validation of the sample adapters compared . . . 67
5.10 Sample application UI flow . . . 69
5.11 Functional areas of the cookbook application . . . 70
5.12 XHTML code for the cookbook's main presentation . . . 71
5.13 Combined action with partial UI update . . . 72
5.14 Data dependent conditional connection . . . 72
5.15 Pure JavaScript and Dojo implementations of the cookbook in comparison 73

6.1 CUI vs. configurable code generator with adapters . . . 77

A.1 Run code generation oAW screenshot . . . 87

List of Tables

3.1 Existing abstract UI description languages in comparison 19

4.1 MARIA XML *only_output* interactors 33
4.2 MARIA XML *interaction* interactors 34
4.3 MARIA XML generic script to JavaScript transformation 40

5.1 Interactor implementations for Dojo and plain JavaScript+XHTML . . . 64
5.2 JavaScript adapters for Dojo and plain JavaScript 65
5.3 Event mapping in the sample application 67

6.1 Sample application comparison for generated and manually developed UIs 78
6.2 Load time comparison for generated and manually developed UIs in ms . 79
6.3 Loading time comparison for generated and manually developed UIs . . . 80