Automated Generation of Abstract Web Applications using QVT Relations

Ali Fatolahi — afato092@site.uottawa.ca
Stéphane S. Somé — ssome@site.uottawa.ca
Timothy C. Lethbridge — tcl@site.uottawa.ca
School of Information Technology and Engineering (SITE) University of Ottawa
800 King Edward, P.O. Box 450, Stn. A Ottawa, Ontario, K1N 6N5, Canada

Abstract. *Web applications development has become a very important track of software engineering due to the rapid growth of the number of web sites. As for other types of software development activities, web development requires effective methods in order to improve the time and cost of the development. The quick rate of technology change is even more challenging to web applications as these applications are accessible online to more users with different interests. In this document, we introduce a model-driven approach that aims at improving the experience of web development for information systems. Our goal is to automate a part of the process. This is the part where platform-independent models are transformed to platform-specific models. In order to minimize the risk of platform change, we generate models that are dependent to an abstract concept of web as an intermediate step toward models dependent to specific web technologies. A developer may eventually transform the output of our method to a specific platform using a platform-specific mapping. Our method generates the domain classes, data access operations and structures, behavior model, presentation model and the flow of the application automatically. The method is also capable of applying changes in reverse order. QVT relations are used to express the transformation rules.*

Keywords. MDA, Web Information Systems, QVT, Automated Transformation.

1. Introduction

Web applications are being widely used nowadays. This requires more effective development methods in order to make the development process faster and less expensive. Further, the quick rate of change in web development technologies highlights the need for methods that generate reusable models, which are independent of target platforms. In this document, a model-driven approach to automate a part of the development process for web information systems is provided. A goal of the approach is to reduce the development time and increase the level of model reusability by helping avoid repeating similar tasks and by providing support for changes.

Our approach takes abstract models that provide a high-level understanding of the behavior and the presentation of the application as input. The output is the design models that describe the behavior, the domain objects, the presentations and the data access mechanisms required for a typical web application. The output could be eventually mapped to explicit platforms to generate an executable application. The input is automatically transformed to the output. We use model-driven development and architecture as a means to reach the goals of this method.

Model-driven development (MDD) is a recent approach that provides means for generating software applications in terms of models and ways of automating parts of the

process. Several approaches to MDD exist; sample sources are (Stahl et al, 2006), (Beydeda, 2005) and (Hruby, 2006). A model-driven development is often performed as a transformation that takes a set of platform-independent models (PIM) and produces a set of platform-specific models (PSM). We break this automated model-driven method in two different steps. The first step takes the platform-independent models and transforms those models to a set of models based on an abstract web model. The second step transforms the abstract web application to a concrete one according to a specified platform.

Model-driven development have been applied to a variety of applications in recent years amongst which, web applications appear to have reached a great level of interest among researchers and practitioners. This is partly due to the fast growing rate of web-based applications. The fast pace of websites development requires great efforts. Web developers are in need of tools and methods to facilitate their work and provide them with more quality development approaches. Model-driven approaches have been excessively used towards web-based development in recent years.

In this document, we describe a method to automatically generate the design models required to specify an abstract web-based application from higher-level behavior models and UI prototypes. This will be done in the context of model-driven architecture (MDA) as defined by OMG (MDA, 2007). We abide by the MDA framework by separating computation-dependent models into two different layers: Platform-Independent Models (PIM) and Platform-Specific Models (PSM). The layer of Computation-Independent Models (CIM) is not currently covered by this method but could be an area of future extension. Figure 1 shows the MDA framework.

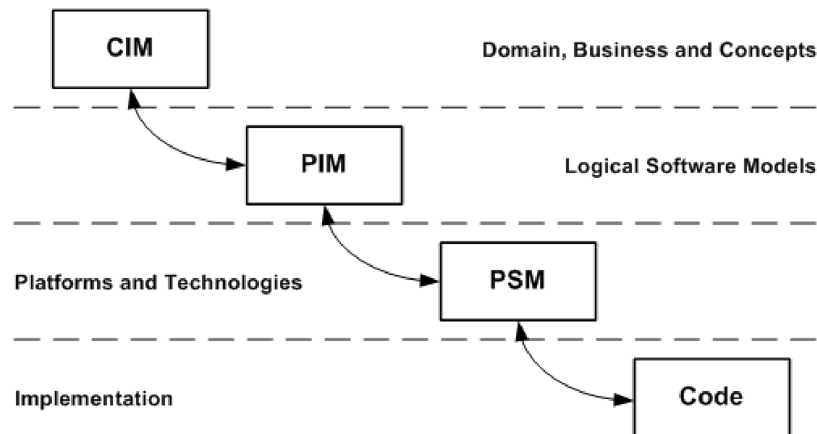


Figure 1 – The MDA Framework

A PIM represents the set of models that define a system regardless of any platform on which the system may rely. In our case, the PIM is defined using domain objects, behavior models represented by state machines and the UI prototypes, which define the GUI layout attached to the presentation states within the state machine.

A PSM deals with the technological details of platforms. The models within the PIM are mapped to certain platforms to conform to a detailed description of the system. In our case two levels of PSMs are pertinent:

1– An abstract PSM (APSM), which is defined upon an abstract model for web-based applications. This is the set of models delimiting universal necessities of web-based applications regardless of the platform to run the application.
2– Specific PSMs (SPSMs): describe concrete web-based platforms such as J2EE or .NET.

The development process aims at the generation of the PSM from the PIM. The traditional approach is essentially a process of adding more details to abstract models in order to reach low-level models that correspond to specific web platforms. Developers starting from a PIM, add design-specific details according to some regulations imposed by a selected implementation platform to generate a PSM. In other words, there is a gap from PIM to PSM, which needs to be filled with platform-specific rules and elements.

The gap between PIM and PSM is to be spanned using a set of transformations. The main objective of our work is to automate such transformations. We approach this goal by decomposing the PIM-to-PSM transformation into sub-transformations from PIM to APSM and from APSM to SPSMs. We will build the PIM-to-APSM transformation as well as examples of APSM-to-SPSM transformations.

Figure 2 depicts a general overview of the proposed method. As this figure shows, the input is provided as state machines representing the behaviors of the system's use cases. The method then maps the input to a default UI model, which may be refined by the developer to build UI prototypes. The dashed line connecting the developer to this step asserts that this step is semi automated.

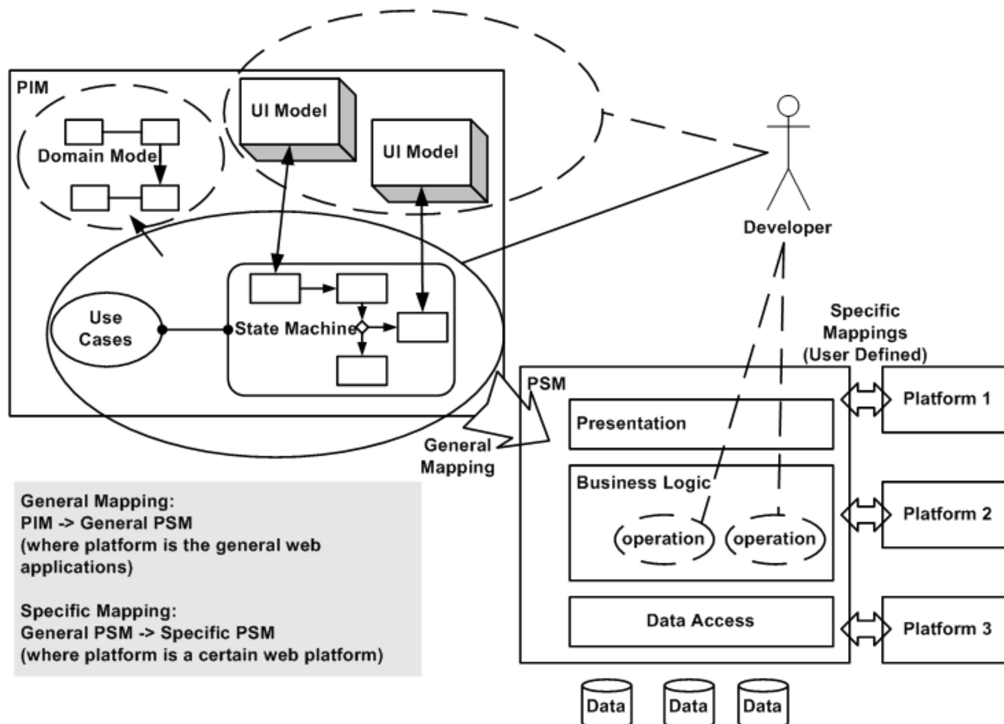


Figure 2 – Overview of the Proposed Method

The state machines and derived UI prototypes are mapped to an abstract model of web-based applications. Meanwhile, the method generates the domain objects

automatically. The developer refines the operations by adding platform-specific code to information-related tasks. One may also opt to apply some changes to the domain model. The method is capable of generating the abstract operations for information processing i.e., creating, reading, updating and deleting data known as CRUD operations. Finally, the abstract web application is mapped to a specific platform chosen by the developer. We use an abstract model introduced by Botterweck (2007) for defining the UI prototype and the abstract web-based application.

The rest of this document is organized as follows. In Section 2, we introduce basic concepts of MDA. In Section 3, we introduce other related work and compare them to our method. In Section 4, we introduce the abstract web model used in this document. In Section 5, we discuss our method in terms of its core mappings. We also provide an overall picture of the approach. Section 6 details the transformation rules and Section 7 contains a case study. Finally, in Section 8 concludes this document.

2. Model Driven Architecture (MDA)

MDA is an effort by the OMG to standardize model driven software development (Stahl et al. 2006). It can be seen as a framework composed of four different layers of modeling. The topmost layer is the layer of Computation-Independent Models (CIM). CIM represents models that are valid in spite of the computational options. Then we have a layer of Platform-Independent Models (PIM). PIMs represent systems and software design and architecture; however, they do not contain any information about specific platforms. The third layer, Platform-specific Models (PSM) deal with the technological details of platforms. Here, logical design models are expressed in terms of specific platforms. At the lowest level, there are Implementation-Specific Models. These are real world objects and components, acting as a running version of the system.

2.1 PIM and PSM

A PIM includes the collection of models that abstract a software system from specific platforms. Use cases and domain models are regular ways of expressing PIMs in the literature. However, the type of model used for describing the behavior of use cases varies. In this document, we accept the following as components of a PIM:

- Use cases are admitted as a part of the PIM but the textual description of use cases is not covered.
- State machines describing the behavior of use cases.
- Domain model is also a part of the PIM. However, our method does not need the domain model to be entered separately. Instead, the transformations build the domain classes automatically. The result is provided to the developer for further refinements.
- User interface templates are considered as a part of the PIM. This is one of the distinguishing features of our method. UI templates specify the desired user interface plus the information elements related to them. The specification of the UI templates must be manually provided by the developer.

In this document, the PSM is generated in accordance with an abstract web platform introduced by Botterweck (2007). The abstract model used includes:

- State machines that are copied from the PIM but enriched with state and transition events, operation calls and parameters that from the behavior of the application as pertinent to the owning use case.

- UI model and components
- Data elements and components used for information storage and retrieval, and to support the UI components
- Domain objects
- Operations used for information processing

2.2. Use Cases

According to OMG (2007), "a use case is the specification of a set of actions performed by a system, which yields an observable result that is typically, of value for one or more actors or other stakeholders of the system". Use cases are means for the documentation of what the system is supposed to do. A use case describes sequences of interactions between actors and a subject system, in order to achieve a goal.

Code Sample 1 is a typical login use case description in natural language text. In this use case, a User enters a username and a password. The system verifies the username and password; if correct, it shows a home page. If the username and/or password are not correct but the User has not reached the maximum unsuccessful login attempts, the system returns to the login page with an error message. But if the maximum number of unsuccessful attempts is reached the account will be locked and the user can select to contact administrator for further assistance, which is a separate use case.

Code Sample 1 – A typical login Use Case

```
Name: login
Actor: User
Precondition: System is up
Steps:
        1- User enters username and password
        2- System verifies username and password
        3- System shows welcome homepage
Alternatives:
        2-a- username and/or password not correct
                2-a-1- System shows an error message
                2-a-2- goto 1
        2-b- login not approved and maximum unsuccessful login attempts reached
                2-b-1 System shows account locked message
                2-b-2- include contact administrator use case
```

A use case model is a collection of use cases, their relationships and the relationships they might have with actors. Actors usually communicate with use cases in order to perform a task. Use cases can include or extend each other. The *include* relationship denotes that a larger use case includes the functionality of another use case. The included use cases could have been fragmented from the larger use cases in order to improve readability or because of the fact that it could be an individual use case by itself that an actor deals with on certain occasions. The *extend* relationship stresses that an extension use case augments the behavior of an original use case. The extension could consist in adding more steps, specializing the way a certain step occurs or providing different responses under alternative conditions.

A use case diagram is used to show the use cases, their relationships and the actors participating in those use cases. Figure 3 shows an example use case diagram. As this figure shows, Customer makes an order in communication with Seller. The *View*

Catalog use case extends the functionality of the *Make Order* use case by giving the option to the Customer to view the catalog. Finally, the *Make Order* use case includes the *Make Payment* use case.

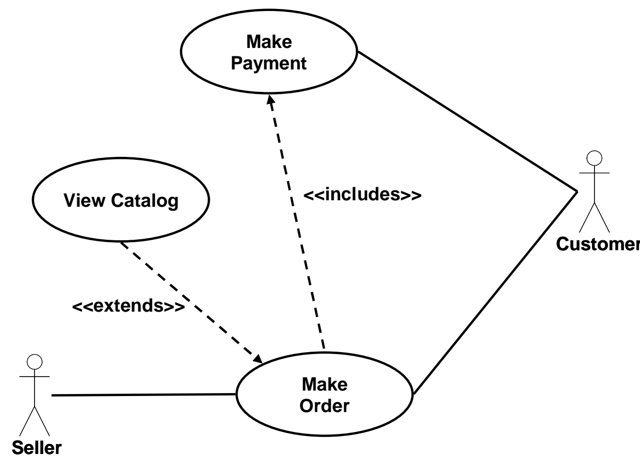


Figure 3 – A Use Case Diagram

2.3. State Machines

We use UML state machines in this document. State machines are used to show the dynamic behavior of things such as use cases, transactions and objects in terms of *states* that are the conditions the objects can be in and the *transitions* that relate two states when an *event* occurs. UML state machines are composed of several elements among which, the followings are pertinent to our work:

- States to represent the status of the web application. A state is usually showing a presentation unit or processing some information
- Choices that are used for building conditional flow of events especially for representing alternative use case steps and use case extensions
- Transitions that cause the change of status because of an event. Transitions in our method carry either events or conditions. The former is usually used to transfer a presentation unit to a processing state and the former is often employed to show a possible value to go out of a choice vertex
- Events that are assigned to either states or transitions.
- Start and end states that if named show a relationship to another use case (state machine), which means that the other use case will run after the successful execution of the current use case.

In our approach, states are generally used for representing different presentation and processing steps. Transitions either represent an event or the result of an operation. The latter is often used when modeling a decision making node using choices. At the PIM level, we only consider states and transitions. Operations, events and parameters will be automatically added at the PSM level. An example is provided in Figure 4, where the state machine describing the behavior of the login use case (Code Sample 1) is presented.

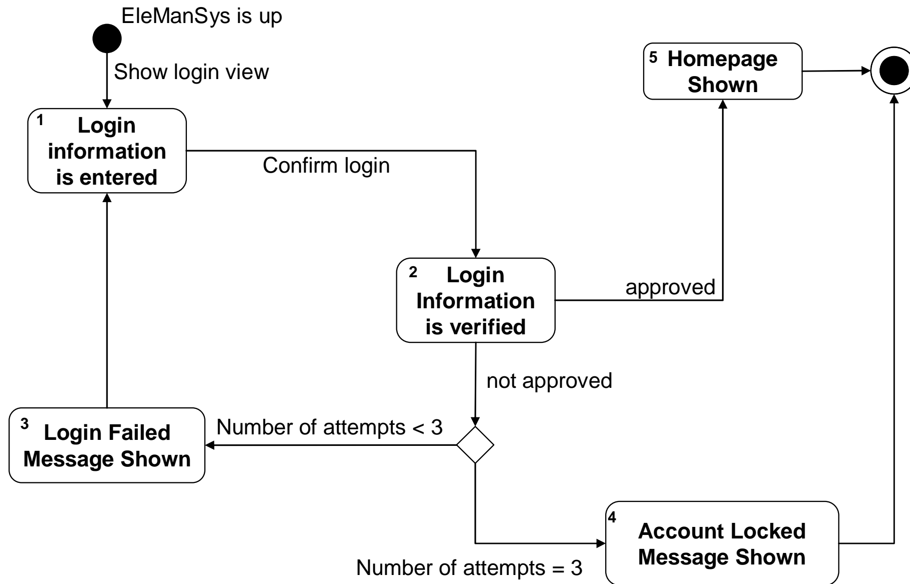


Figure 4 - login state machine

2.4. Mappings, Transformations and Relations

A key aspect of MDA is the capability of model transformation. As Pierantonio et al (2007) put it; transformations are the chaining feature that makes the automation process possible. OMG (2003) defines a mapping as a specification describing how to transform a PIM to PSM, although the same definition could be used for other types of mappings. This does essentially mean that a mapping is a platform-specific notion – according to OMG (2003) - and the characteristics of a mapping are decided according to selected platform(s). Correspondingly a transformation is defined as an execution of a mapping by in MDA. We will use QVT relations as one MDA standard for formalizing transformations. Code Sample 2 shows the syntax of a QVT Relation.

Code Sample 2 - QVT Relations Syntax (OMG, 2005)

```
Relation R
{
        Var <R_variable_set>

        [checkonly | enforce] Domain:<typed_model_1><domain_1_variable_set>
        {
                <domain_1_pattern> [<domain_1_condition>]
        }

        [checkonly | enforce] Domain:< typed_model_n>
        <domain_n_variable_set>
        {
        <domain_n_pattern> [<domain_n_condition>]
        }

        [when <when_variable_set> <when_condition>]

        [where <where_condition>]
}
```

Code Sample 2 defines the syntax of a relation *R* that transforms *typed_model_1* to *typed_model_2*, referred to as a source and target domains respectively. This naming convention, however, does not impose a direction to the transformation. The direction of the transformation is decided based on the keywords *checkonly* and as below:

- if none is used the domain is considered *enforce* by default, which means that this domain will indeed change as the transformation happens. In other words, an *enforce* domain is always a target domain.
- if domain 1 is tagged as *checkonly* and domain 2 as *enforce*, then the transformation is a unidirectional transformation from source to target. This is the most common way to do transformations. In this case any change in the target model will only result in a check message towards the source model
- If both domains are marked by *enforce* or left unmarked, the transformation is bidirectional.

In Code Sample 2, the *when* clause indicates a pre-condition that must hold before the matching of the source and target. Respectively, the *where* clause states a post-condition. The patterns used for defining source and target domain are by themselves some sort of preconditions. In order to have a transformation from *a* to *b* validated, the pattern defining domain *a* must be found within the source domain.

A QVT relation could be a top relation; that is a relation that the transformation starts with. Non-top relations could only be called within other relations as a part of their *when* or *where* sections. Code Sample 3 shows an example of *ClassToTable* relation from QVT official document.

Code Sample 3 - Class to Table QVT relation (OMG, 2005)

```
top relation ClassToTable // map each persistent class to a table
{
	cn, prefix: String;

	checkonly domain uml c:Class {namespace=p:Package {},
								kind='Persistent', name=cn};

	enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,column=cl:Column
							{name=cn+'_tid', type='NUMBER'},
								key=k:Key {name=cn+'_pk', column=cl}};

	when {
		PackageToSchema(p, s);
	}

	where {
		prefix = '';
		AttributeToColumn(c, t, prefix);
	}
}
```

Code Sample 3 describes a relation mapping a class to a table. The transformation happens from classes to tables and not the other way around. In order for the transformation to be valid, it is necessary that the *PackageToSchema* relation hold. PackageToSchema relation maps a package to a database schema; this means that the table must belong to the schema generated as the result of mapping the owning package

of the source class. The object pattern defined for the domain *c* indicates that the transformation looks for classes within the source model that are *persistent* and have a name. The transformation will then map every single of such classes to database tables with the same name as source classes and a default column as the primary key. Afterwards, the transformation will call another relation to map the attributes of the class to columns of the table.

3. Related Works

We study the published work informally describing how to develop a PSM for web applications in a model driven sense, in order to assess what are the important aspects to consider. Since we are using other tools for validation and implementation, methods using existing tools are also considered as related. Models and methods that are formally defined for model driven web-based development form another group of the related work.

A group of researchers have provided their methods in terms of a set of guidelines that steer developers throughout the process of achieving the PSM of web-based applications from the PIM. Such approaches are not automated but show how it is possible to reach the PSM of web-based applications from the PIM. For example, Li et al. (2000) illustrate a UML-based approach to devise the architecture of web-based applications. Meliá and Gómez (2005) provide an MDA-based approach supported by QVT rules to derive an architecture-centric web-based application. Their method is based on merging functional and architectural viewpoints to reach the PIM. It is based on the so called MVC2 pattern (Meliá et al, 2005) for web applications.

We are especially interested in model-driven approaches to web-based development. Model driven architecture and other model driven approaches have a rich arsenal of models and formalisms required to build up the basics of automated methods for development purposes. This ability has prompted many scholars and practitioners to use model driven approaches to automate the process of developing web-based applications.

Some of the existing model driven methods are supported by tools that come with a user-friendly suite, documents and tutorials. For example, WebML (2008) was introduced by Ceri, et al (2000) and is supported by WebRatio (WebRatio, 2008). The method is now enhanced to adapt with model driven methods according to Brambilla et al. (2007) and Ceri et al (2006). WebRatio is one of the platforms that we plan to use to validate our approach when transforming the APSM to SPSMs. The tool may provide a way to show that our method can generate appropriate APSM that could be transformed to different platforms. Another example is a MVC-based approach to generate J2EE applications using eclipse-based plug-ins proposed by Tai et al. (2004). Meliá et al (2007) report the usage of WebTE, which is a tool supporting WebSA (Meliá and Gómez, 2006) method for model-driven web development focused on configuration management and software architecture. Muller et al. (2005) present an experiment with Netsilon (2008) to generate abstract web application models and transform them to specific platforms. This work is close to our work and will be discussed in more details.

A number of model driven methods are based on previously developed modeling profiles. For example, UML-based Web Engineering (UWE) is a UML-based profile for web-based applications (Kroiß and Koch, 2008). A method to use UWE is reported by

Koch et al. (2001). The whole method is based on a PhD thesis by Koch (2000) for which José et al (2006) have eventually prescribed a method to gather requirements. A number of model-driven approaches for the semi-automated generation of web-based applications using UWE is introduced by Kraus et al (2007), Koch et al (2006) and Koch and Kraus (2002). One more work reported based on UWE is done by Cáceres et al (2006). Another example is described by Rossi and Schwabe (2006), which is an experiment with the Object-Oriented Hypermedia Design Method (OOHDM) to develop model-based web applications. OOHDM is an object-oriented framework to define web-based applications. Another work based on OOHDM is published by Schmid and Donnerhak (2005). They describe a PIM-to-PSM transformation to map an OOHDM-based model to a servlet-based platform. The approach results in two different PSMs: conceptual and navigational. However, the implemented method only covers the navigational aspects. Navigations are separated as those to fixed and dynamic pages. A semi-formalized language is used to generate the PSM. Ubiquitous Web Application (UWA) design framework is a user-centered set of tools and techniques for designing web applications (UWA, 2002). Distante et al (2007) use UWA along with MVC in the context of model-driven development. They define a UML-based MVC pattern and map it Java Server Faces (JSFs) to generate the GUI of web applications. Another similar method is described by Wu et al (2007) to generate JSP pages for a web application from use cases by applying robustness analysis.

From the viewpoint of our method the following points are important and have not been visited properly by other works:

- Most of other approaches focus on the presentation, navigation and hypermedia content of web-based applications, while our research focuses on building the application itself with a stress on the objects required to connect different layers and the operation required to run the application services.
- Even the papers addressing data-centric or information-based web applications - such as (Ceri et al, 2006), (Nikolaidou and Anagnotopoulos, 2005) and (Baresi et al, 2006) do not tend to cover the data access layer; rather they focus on the presentation and navigation required to represent this data and information at the hypermedia level. Again our research considers the data access layer and the database itself.
- Introducing the notion of APSM and SPSMs is another distinguishing aspect of our approach. Transformations are usually defined towards a PSM. Exceptions such as He et al’s approach (2005) and UWA (2002) end up with models that are very general and too abstract. As a result, most of the research lead to approaches that are either hard to map to specific platforms or hard to adapt with other platforms. Our approach distinguishes APSM and SPSM so that the method could be flexible in terms of adaptability with web platforms.
- The exact semantics of the transformations required to automate the process is rarely given in the literature. Instead, there are general rules or guidelines describing how to do the transformation automatically or how to build and configure the environment to do so. In a few cases, a subset of transformation rules is described in more details but not in a formal way, examples are (Kraus et al, 2007) and Muller et al (2005).
- The transformation provided in our method is defined using a bidirectional mapping. Most other approaches fail to address this issue resulting in hard-to-maintain environments. We deal with this problem using QVT relations without

checkonly/enforce prefix making it possible that any change in the target model can be applied to the source model and vice versa. This method of transformation needs careful attention because the applicability of every relation needs to be ascertained in both directions.

- Finally, we see the UI prototype as a requirement that should be defined as an input while some other approaches try to generate the UI as an output, e.g. (Wu et al, 2007). Our position is that the desired UI model should be seen as a requirement. This also includes the usage of information elements used in the description of UI prototype as well as the query marks used to elaborate the relationships between a UI element and associated information elements that are used for the automated generation of data-related operations.

Amongst the mentioned related work, there are approaches that have similarities with ours but none has all the characteristics that we propose. For example, Wu et al's method has a set of detailed transformation rules but it only covers the presentation layer. UWE-based approaches result in an abstract web-specific model that could be mapped to any platform but they do not provide detailed transformation rules and they are not bi-directional. Instead they focus on the UWE meta-model and how to use it. This is also true about the approaches using OOHDM. Automation is also a feature missing from most of the related work.

Table 1 lists the features that are important from the viewpoint of our method and compares it to the most related work, which are the work that provide model-driven approaches to web-based development. One can see from Table 3 that our approach has unique features such as the support for bidirectional mappings and generalization mechanisms through abstract PSM and specific PSMs. When it comes to automation, it is very hard to assess the existing work as they all claim to be automated. However, due to the following facts those approaches appear to be providing tools and models required for building automated methods rather than being automated by themselves:

- lack of details on the automation
- unclear separation of manual and automated parts
- implementation mechanism is not discussed for so called general approaches only specific instances leading to concrete platforms are elaborated

One last note regarding the data modeling and data access mechanisms in the related work is that almost all the similar works reported in Table 3 cover the topic of data modeling and navigation modeling. However, data modeling is different than data access mechanism. The latter is missing from most of the related work. Modeling the data access through the automated generation of classes and operations required to retrieve and store data is another major concern of our method.

Compared to other related works in Table 3, our approach has two main limitations. One is that the coverage is only given to the PIM-to-APSM step, while most of other related works support a larger scope. The second limitation is the fact that we only approach web-based information systems while other works tend to be more general. However, limiting the scope as we do in this document, has resulted in an approach that is fully-automated and covers the automated generation of data access mechanisms that are missing from other methods

Table 1 - comparative study of the most related work

	UWE-based methods	Rossi and Schwabe (2006)	Schmid and Donnerhak (2005)	Wu et al (2007)	Distante et al (2007)	WebML-based work	WebSA-based work	Muller et al. (2005)	Our approach
Covered Layers	All	All covered but the data access mechanism is not specified	Presentation and navigational aspects	Presentation	Presentation and domain model	all	all	all	all
Format of transformations	ATL rules	XSLT specification for widget elements only mapped to navigation and concrete UI elements	Semi-formalized text	Textual rules	Informal textual rules	XSLT used but not presented	XSLT used bot not presented	Not specified	QVT Relations
Direction of transformation	unidirectional	unidirectional	unidirectional	unidirectional	unidirectional	unidirectional	unidirectional	unidirectional	bidirectional
UI as a requirement	yes	yes	no	no	no	yes	no	yes	yes
Target platform	Specific examples provided	OOHDM-based abstract	servlet-based	JSP-based websites	JSF (Java Server Faces)	Java	Java	Java, PHP	Abstract Web
Generalization mechanism	Manual customization of the method required for every platform	Not specified	none	none	Not specified	Not specified	Not specified	Separation of platform-dependent from technology-dependent PSMs	Separation of PSM to APSM and SPMSs
Scope of Software Engineering	Code generation not included	Code generation exempted	Requirements Engineering not included	Requirements Engineering not included	All	All	All	all	Code and requirements not included

4. Abstract Web Model

The specification of a mapping requires the specification of the source and target models. Since the mapping discussed in this document happens in the area of web-based information systems, the reference models used must capture:

- the abstract structure of a web-based application preferably in accordance with MVC
- the abstract UI model
- the abstract data model

We use the model introduced by Botterweck (2007). This model supports the data model, UI model and state-machine-based behaviors for web-based applications. It adapts with UML and a mapping to UML exists. The model will be used as the reference model for both the source and target models with a few considerations. This model includes the following packages:

- *State Machine* includes the elements required to build a state machine in accordance with UML state machines.
- *UI Structure* encloses the elements required to build the general structure of the UI model such as pages, units and navigation links.
- *UI Components* includes the modeling elements for the UI components used for communications with end users.
- *Data Model* follows UML core model for specifying classes, objects, data types and their attributes.
- *Data Components* is a part of the model that relates the data model to UI components in order to transfer data in and out of the presentation layer.
- *Web Services* is a part of the model that is provided to give support to web service applications. The package can be used for regular web applications as well. It allows some functionality to be introduced as services. The developer can then choose to deploy them as real web services or to implement as internal services. In the later case, because the functionalities are tagged or stereotyped as services, it is always possible to eventually change them to web services.

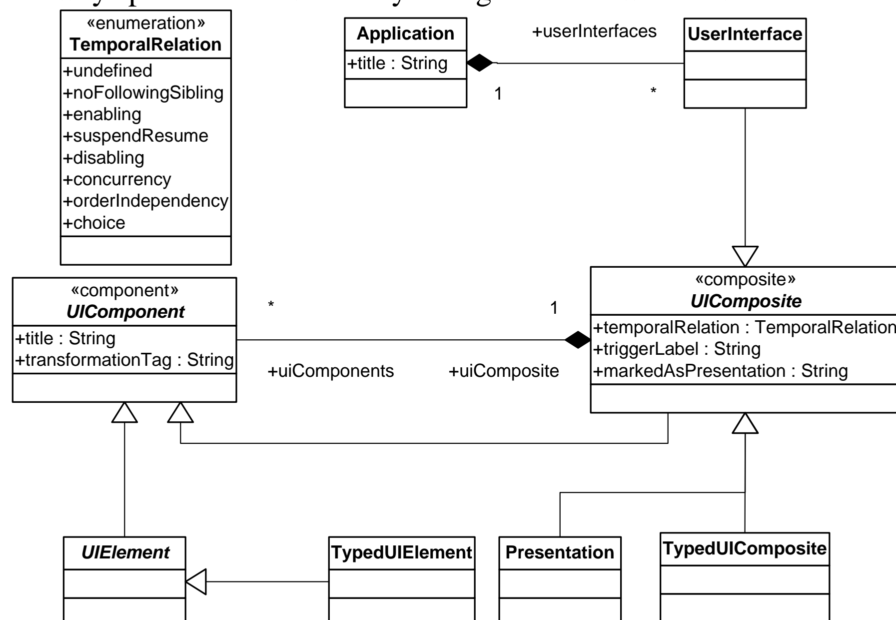


Figure 5 - UI Structure Model (Botterweck, 2007)

In this section, we only review those parts of the model that are required to describe mappings and transformations.

Figure 5 details the UI Structure package. According to this model, an application can have several user interfaces. A user interface is in fact a special kind of UI composite, and a UI composite is a collection of UI components. UI components are elements that the user can see and directly communicate with. A UI composite may be a presentation, which in association with a state makes a presentation state meaningful. This package also contains the abstract elements used by concrete UI elements in a UI element model.

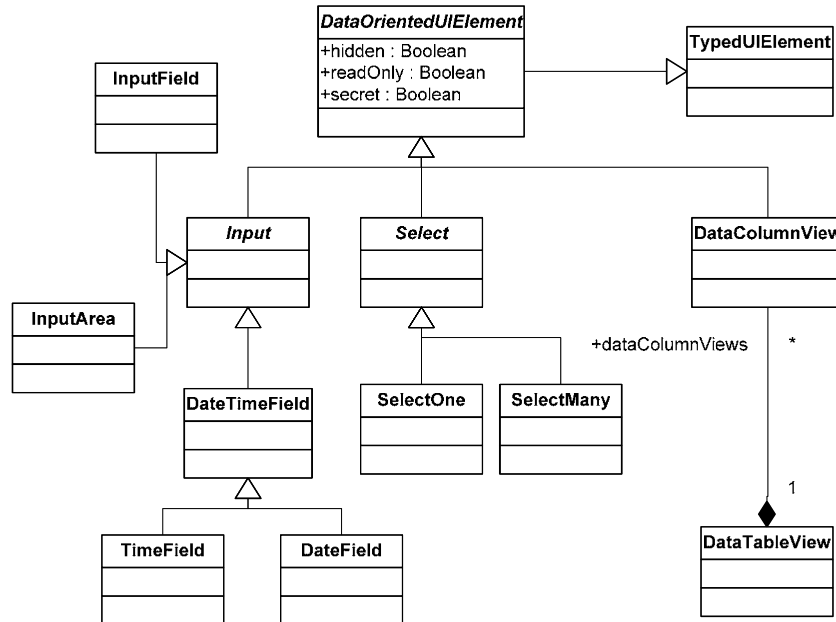


Figure 6 - Data-oriented elements of UI components (Botterweck, 2007)

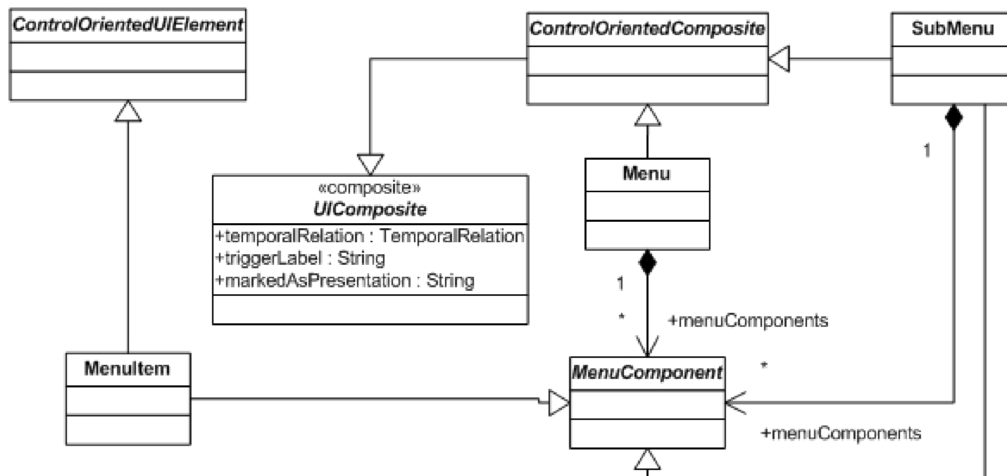


Figure 7 - Menu items (Botterweck, 2007)

Figure 6 shows a subsection of the UI components model. These components include: components connected to data elements, and composite elements such as input fields, selection boxes and table views. Figure 7 shows the elements required to model

menu items. Figure 8 presents the model for plain hypertext, links and buttons. Triggers occurring in a presentation could be of one of the following three types: hyperlinks, operation triggers such as submit buttons that require an action in the controller and/or service layer and navigation trigger that only cause a change in navigation path.

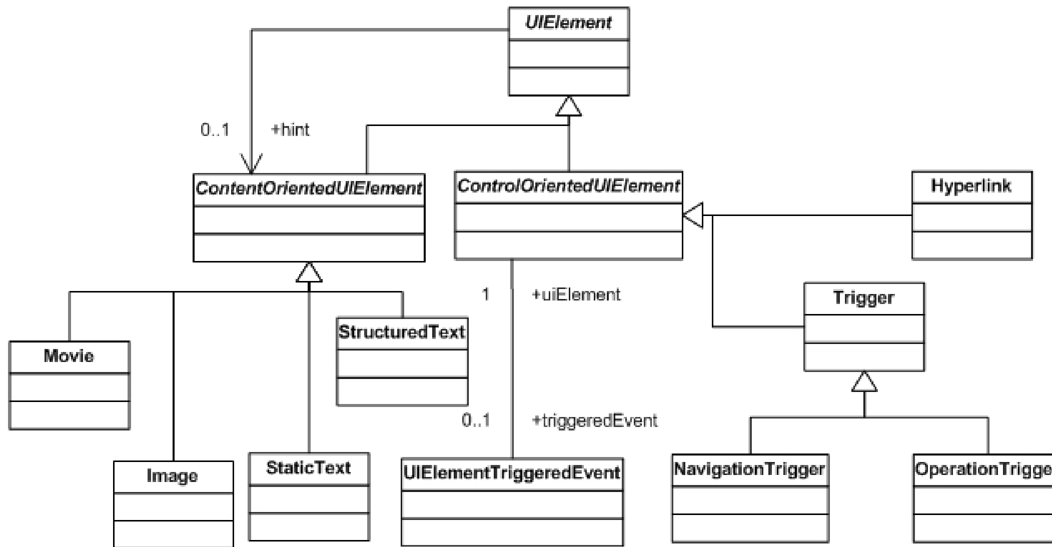


Figure 8 – Content and Control Oriented Component (Botterweck, 2007)

We added a new concept to the UI model to support the required data for UI components. Our assumption is that a database already exists so the developer is required to know what data is associated with each component. Alternatively the developer may choose to manually add modeling elements required to support a component. This feature is not supported in the original abstract model to the required extent. Some data features can be associated with input fields such as select lists but not to other elements. We require the possibility to associate data elements to operation triggers in order to be able to automate the process of generating data-related operations and services.

We introduced the following format to associate UI components with data : *<UIComponent><DataOperation><DataSource>*, where *DataOperation* is one of the followings:

- ? means that the contents of the form will be matched against the data from the *DataSource* as a selective query
- >> updates the *DataSource* with form values
- << loads all elements from the *DataSource*
- + adds the form values as a new entry to *DataSource*
- - removes an entry from *DataSource*

We customized the reference model with *OperationTrigger* associated to *DataElement* as shown in Figure 9, in order to support these elements.

Figure 9 - OperationTrigger can access DataElement to build a queryData link

We also made navigation from an operation trigger to its owning presentation possible. An operation trigger, which is a specific type of a UI Component, cannot access its owning presentation, which is a specific type of the UI Composite. This is resolved in Figure 10, where the navigation from UI Component to UI Composite is made possible with an additional association.

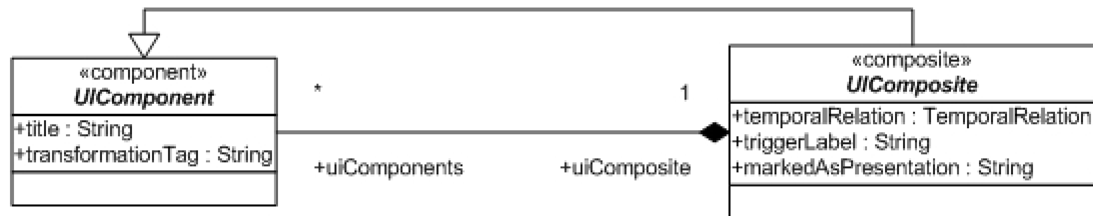


Figure 10 – Navigation from UI Component to owning UI Composite made possible

Another change was required to denote the default event on a presentation unit. Presentation units could contain several kinds of events that are not necessarily operation triggers (i.e. submit button). For example, drop-down select components can cause a selection event that affects the contents of other components. In order to enable the description of such behaviors, we have added a new association between *DataComposite* and *UIComponent* elements. This association is shown in Figure 11.

Following is a list of important associations that exist on other parts of the abstract model that are required for better understanding of mappings and transformations:

- A *transition* can be associated to several *Operation*s. This capability is used to model controller operations (Figure 12).
- Transitions can also be associated to several *Events*. This serves to model signal events over the transitions. UI triggered events are a special type of events, which can have *operation triggers* as triggers (Figure 12).
- Data Oriented elements are associated with Data Composites that are collections of data elements. Data composites are connected with a special class called *OperationAdapater*, which could be assigned one of the following roles corresponding to CRUD operations (Figure 13):
  - selectOperation
  - insertOperation
  - deleteOperation
  - updateOperation
- A data element has an association to Classifier denoting its type (Figure 14).

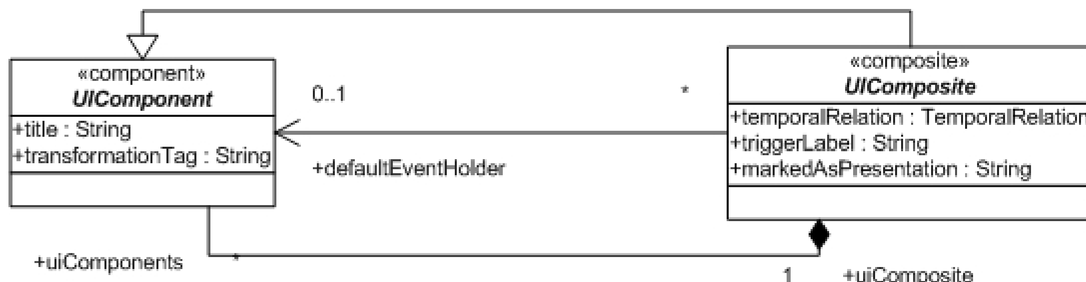


Figure 11 - *UIComponent* as a default event carrier for *UIComposite*

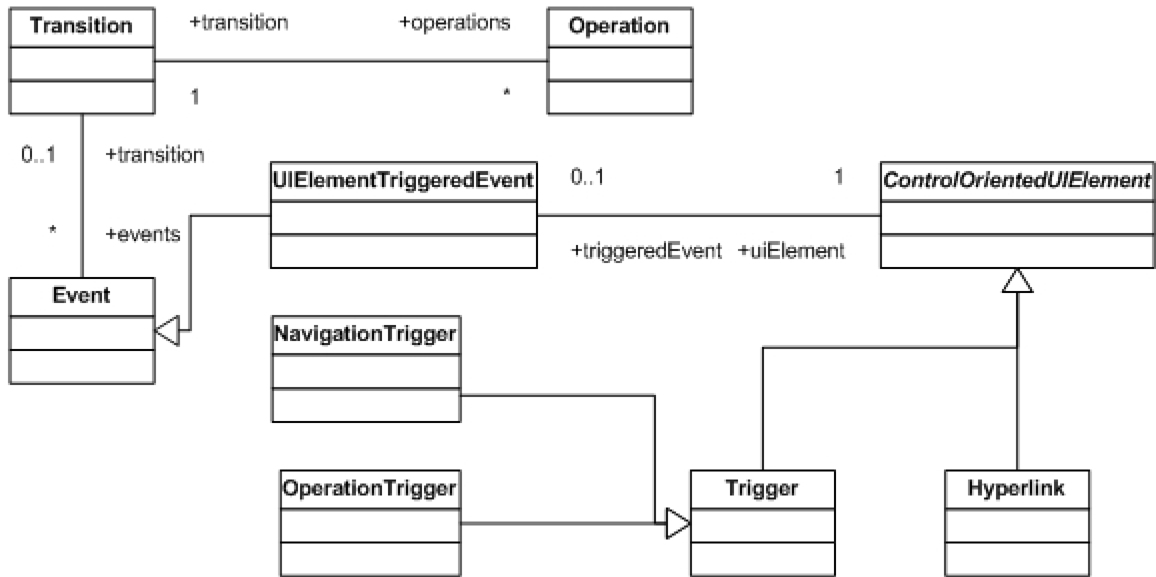


Figure 12 - Transition in relation with events and operations (Botterweck, 2007)

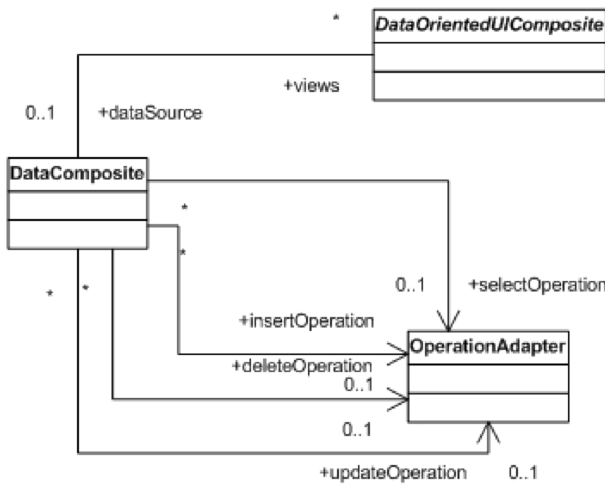


Figure 13 - Data Composite related to Operation Adapters (Botterweck, 2007)

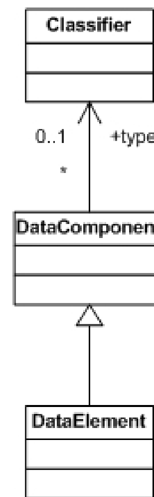


Figure 14 - Type of a Data Element (Botterweck, 2007)

5. Mapping from PIM to PSM

In this section, we will describe our proposed method in terms of the mappings and transformations defined so far. A big picture of the whole mapping for better understanding is provided in this section but the details of transformations are provided in a separate section.

Figure 15 provides an overall view of our proposed method in terms of mappings that transform PIM to APSM. The developer defines the state machine in terms of states and transitions, and a UI model attached to every state including data sources related to every component. The method is defined in terms of the transformations that automatically map the PIM to the abstract-web specific models. The result of mapping includes the following:

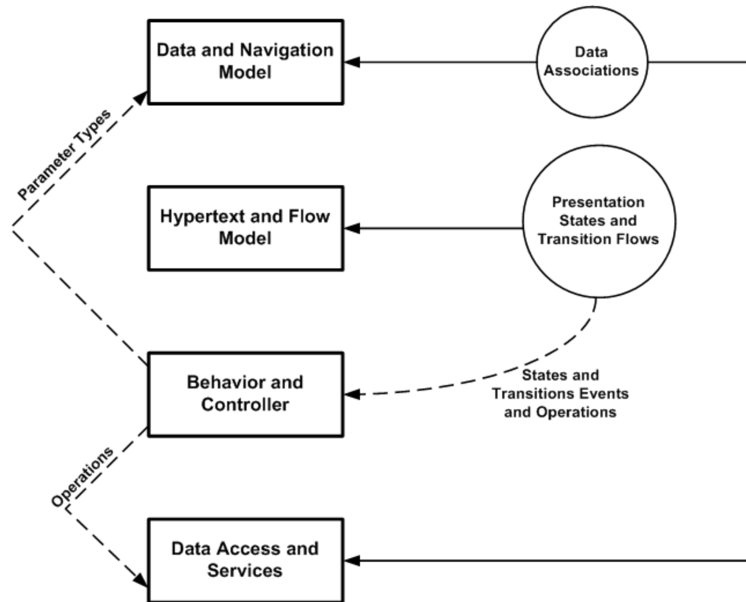


Figure 15 - A Big Picture of the Mappings

- different types of signal/call events that are required to steer the transitions through different web pages
- the domain objects and the navigation model that describes the type of associations existing amongst those objects
- other objects that act as a collection of objects required to manage information
- operations required to access data and control behavior
- copy of state machines and UI models

The transformation process described by Figure 15 is as follow:

- Data objects and navigations through data objects are created based on data associations found within the UI model.
- The contents of the presentation states and transitions flows end up with the content and structure of as the hypertext level in terms of UI components, links and pages.
- Transitions and states from the input model, are also used to create events and operations used for the generation of the behavior and controllers of the application.
- The generated behavior and controllers is used in turn to build the data access services in combination with data associations from presentation states. It is also used to map parameters to attributes within the data model.

A web-based application is composed of a set of web pages. Each web page may contain one or more presentation units. Presentation units act as either input or output units. Input units present forms and other means of entering data to the user and eventually use those data for processing. In the case of information systems, such processing usually involves database operations. Output units are used for providing results or information to the user. It does not matter if a page is composed of several units because the method is based on the presentations (units) and not the web pages.

Figure 16 shows a simple state and the result of transforming this state. This figure presents a big picture of our method and hides details regarding the creation of

parameters and associations. This figure shows that the input fields are mapped to signal events and the data associated with the submit button is mapped to both a domain object and an operation that performs the required action. Since the method generates an abstract web application, it is not possible to generate operation codes. We only specify the signature of operations. This is helpful enough to map operations to suitable data access code according to selected implementation platforms. In Figure 16, the left column contains the PIM and the right column shows the generated APSM. Table1 and Table2 are the data sources generated in the APSM as a result of transforming Data1 and Data2 respectively from the PIM. We use the notations within Table 2 for associating data element to UI elements.

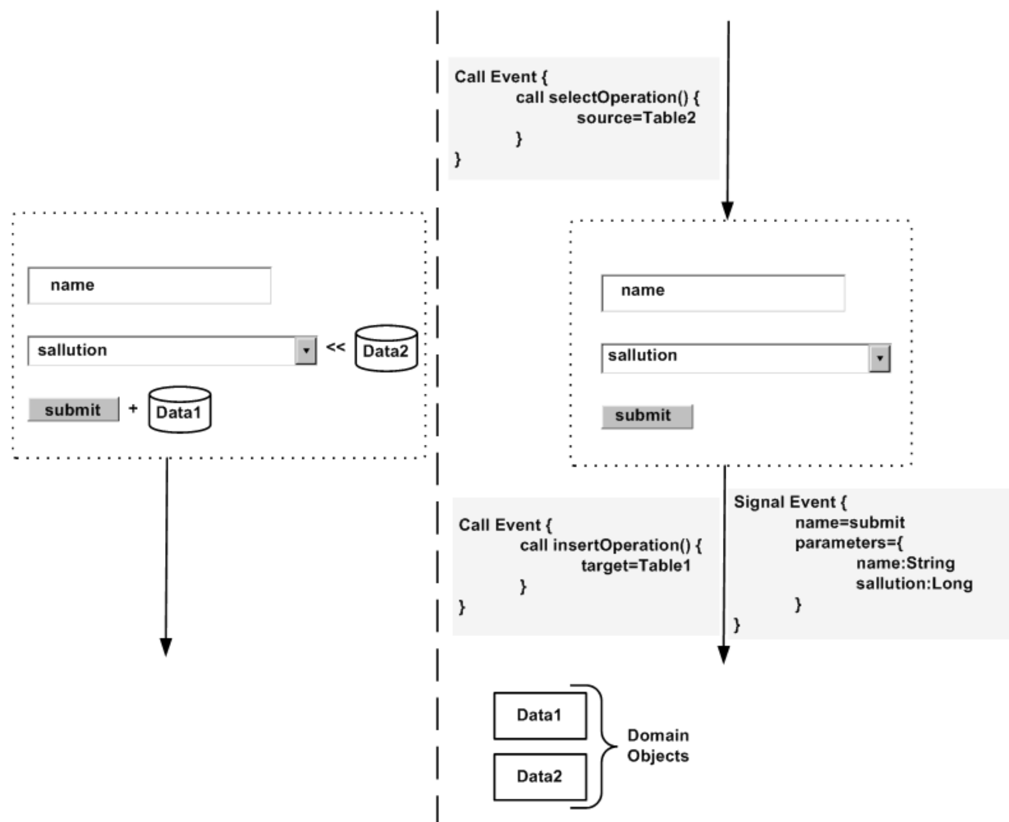
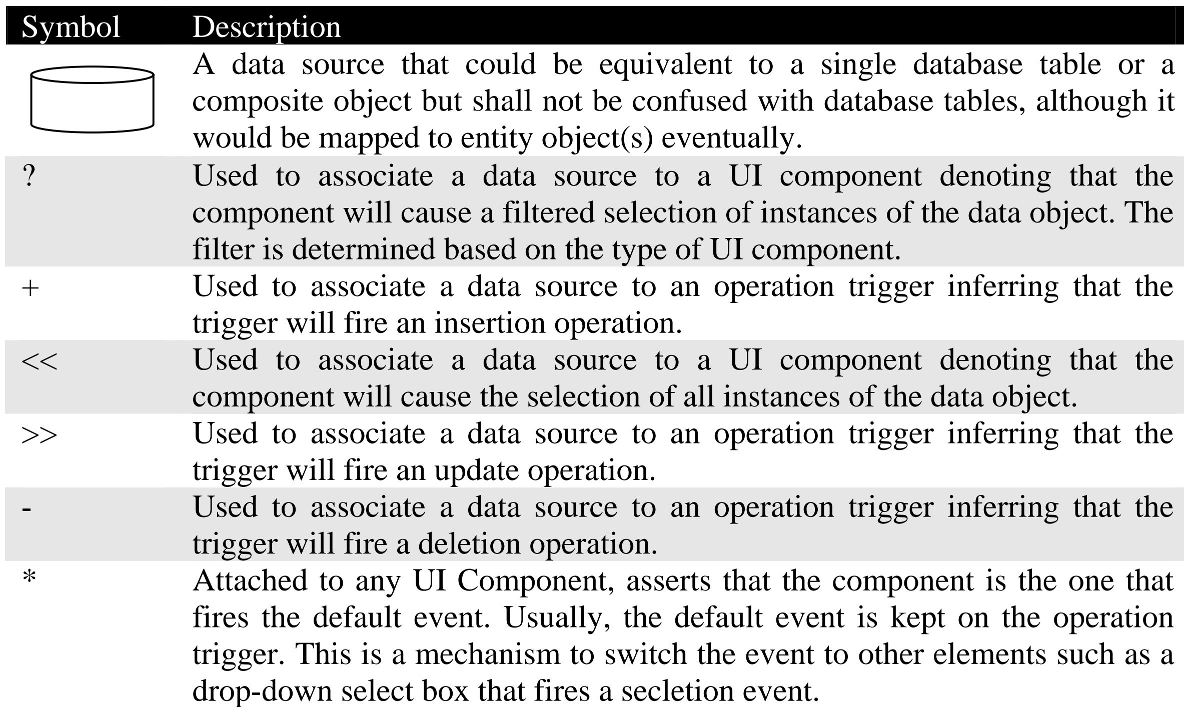


Figure 16 – Mapping a sample presentation state to presentation, behavior and data

Following is a typical process for our approach:

1. The developer models the requirements using state machines and UI prototypes
2. System generate the APSM in on automated step (which is the contribution of this method)
3. The developer selects a specific platform
4. if the corresponding APSM-to-SPSM mapping is found among the default mappings
   4.1. System generates the SPSM
5. else
   5.1. the developer defines the APSM-to-SPSM mapping
   5.2. goto 4.1
6. The developer transfers the SPSM to the corresponding tools

Table 2 – Graphical Symbols Used for Associating Data with UI Model



Symbol	Description
	A data source that could be equivalent to a single database table or a composite object but shall not be confused with database tables, although it would be mapped to entity object(s) eventually.
?	Used to associate a data source to a UI component denoting that the component will cause a filtered selection of instances of the data object. The filter is determined based on the type of UI component.
+	Used to associate a data source to an operation trigger inferring that the trigger will fire an insertion operation.
<<	Used to associate a data source to a UI component denoting that the component will cause the selection of all instances of the data object.
>>	Used to associate a data source to an operation trigger inferring that the trigger will fire an update operation.
-	Used to associate a data source to an operation trigger inferring that the trigger will fire a deletion operation.
*	Attached to any UI Component, asserts that the component is the one that fires the default event. Usually, the default event is kept on the operation trigger. This is a mechanism to switch the event to other elements such as a drop-down select box that fires a secletion event.

5.1. Signal Events

A signal event acknowledges the receipt of an asynchronous message. Signal events are added to transitions. In a web based application a signal event represents a request by a page for the execution of a process. The best examples of such events are events fired by submit buttons. In order to generate signal events, every presentation state is examined to verify if it has an operation trigger. The signal event will be created with the same name as the operation trigger on the outgoing transition. Parameters to be submitted to the signal event are created according to the set of input fields attached to the operation trigger. In a regular web application, these input fields are the form fields of a submit action. Following is the details of this mapping:

- **name**, the intact copy of the name of operation trigger
- **transition**, an outgoing transition going out of the presentation state holding the operation trigger
- **parameters**, one parameter will be created per each input filed found within the operation trigger. The type of parameter is selected according to a mapping function that maps UI input fields to signal event parameters as follows:
  - select input fields are mapped to *Long* data type because it is assumed that the actual data the developer needs to supply a select component is the id of the data elements. Multiple select inputs are mapped to an array or collection of *Long* inputs.
  - check/uncheck fields are mapped to *boolean* fields.
  - table fields are mapped to a collection of the specified object
  - other fields are mapped simply according to their type. For example, simple text inputs are mapped to *String* data type. *Date* and *time* fields are mapped accordingly to data and time data types.

An event does not always relate to an operation trigger. It often happens that a change in the status of a component other than an operation trigger results in changes to other components in the same page. Figure 18 shows an example of such situation. In this example, the selection of a province in the first state results in the population of the list of towns and villages. In such cases, the mapping will use the information from the default event holder of the presentation as elaborated in Figure 18, to generate the signal event. When no default event holder is defined, the submit button is taken for that purpose.

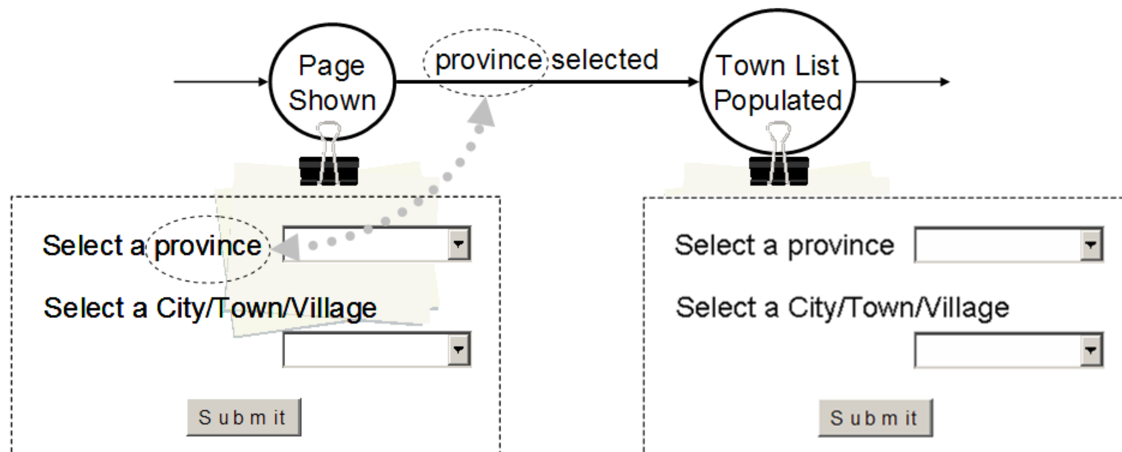


Figure 18 – Signal Event created based on the selection event of a drop-down list

The details of the generated signal event are as follows:

- **name**, a copy of the name of the component, data association and its type. For example, a select component associated with the data object, *Province* will result in the creation of a signal event named *selectProvince*.
- **transition**, an outgoing transition going out of the presentation state holding the component.
- **parameters**, one parameter will be created according to a mapping function that maps UI input fields to signal event parameters as described already. For example, in case of Figure 18 the signal event will carry a parameter, *province* of type *Long*.

5.2. Call Events

Call Events are added to states or transitions. Call Events confirm a call to an operation. We refer calls from call events to use case controllers. A call event is created on a state if an incoming transition to that state carries a signal event. The event calls a controller operation, which has the exact same parameters as the signal event. The details of the mapping are as follows:

- Operation parameters are copied from the signal event.
- The type of operation to be called from a data service depends on the type of data association of the corresponding operation trigger with a data source. The data source is used to generate the data object, which is in turn used to define the type of returning or input parameters of the operation. Details are as follows:
  - ?: a query operation that compares a collection of data objects with the parameters entered in a corresponding operation trigger. An example is a login operation that

looks up a username/password pair in a users database. This operation would return a boolean value.
  - +: insert operation to add a new instance of an object.
  - <<: a query operations that returns all instances of a data object.
  - >>: an update operation that updates an instance object using the input parameters.
  - -: remove operation that deletes an object instance.

5.3. Controllers and Services

In the MVC architecture, a controller class is usually a class introduced to control an application based on the behavior defined by a use case or a similar unit of behavior. In addition to controller classes, one or more service classes are used to perform the operations required to access data services. Different platforms may use different implementation strategies for the controller and data access service classes. We provide the minimum that is required for decision making when mapping to specific platforms. In order to do this, we use the notion of *OperationAdapter* from our reference model. Operation Adapters are used for building the required operation based on the data association type used in the UI model. There are four types of operation adapters listed as *selectOperation, insertOperation, updateOperation* and *deleteOperation* representing the four main categories of database related operations.

Every presentation corresponds to a controller. Each controller has a corresponding operation adapter for each data element that it needs access to. An operation adapter may implement a number of operations according to the type of action imposed by the corresponding call event. Operation adapters represent the service operations that controller operations call. However, the mechanism to generate the controller operations is different.

We also create an operation for each transition ending to a choice. A natural language description is produced as pseudocode for that operation. This pseudocode description is created according to the name of the operation trigger, the type of the data association used with the operation trigger and the name of transitions going out of the target choice.

For example consider Figure 16 and suppose there is a choice after the state, Code Sample 3 shows the pseudocode for the corresponding controller operation.

Code Sample 3 – Example pseudocode added to controller operation

```
– if (name, sallution) was added successfully to Data1 return true else return false

– call insertData1(name, sallution) service for this purpose
```

The return type is defined based on the state preceding the transition. If the transition comes out of a presentation state with an operation trigger attached to a data source the return type is *Boolean*. The returned value indicates if the data operation was successful. For other cases, the returned type is *Integer* in order to be able to handle more than two possible outcomes. In such cases the calling line corresponding to the last line in Code Sample 3 is absent.

Controllers operations are also generated to supply data to data-oriented components within the presentation. The name of such controller operations depend on

the type of the component and the associated data. For drop down select boxes, the controller operation is named according to pattern *populate* + *<componentName>*. For example, Code Sample 4 describes the controller operation used for populating the list of provinces in Figure 18.

Code Sample 4 – Pseudocode added to a controller operation for populating a drop-down component

```
PopulateProvince() controller operation

– Call selectProvinces():Provinces

– Remarks: Provinces is an array or collection of Province
```

5.4. Domain Objects and Navigation Model

Domain objects are mainly discovered from the *queryData* role introduced in Figure 10 and from other existing data element associations supported by the abstract model in relation to UI components. In order to determine the type of the domain objects, it is important to understand the type of the element holding an association with it. We use the term *holder* to refer to this element. The respective data elements are mapped to classes. The attributes of these classes are determined according to the following:

- If the holder is an operation trigger, the UI components referred to in the operation trigger are used as the source of a mapping targeting the attributes. Attribute types are defined using another mapping function that generates an attribute type based on each UI component.
- If the holder is a table, the columns of this table are used as attributes. In practice, most of the tables found on web pages are in fact the result of composing more than one object but the composite object itself is not a member of the database. We do not necessarily deal with concrete database objects but with data objects. These data objects may be composite or single. The developer shall use a composite data element symbol in the case that composite objects are used. Otherwise the method would only build the basic operations required to handle the basic data elements and not the composite ones. This does not result in a failure but require more subsequent refinement by the developer.
- For other types of UI components the mapping results in a class with no attributes. Notice that since the mapping considers all possible situations, it is possible to eventually map attributes found using another component such as an operation trigger to an object created as the result of a mapping from a component without attributes.

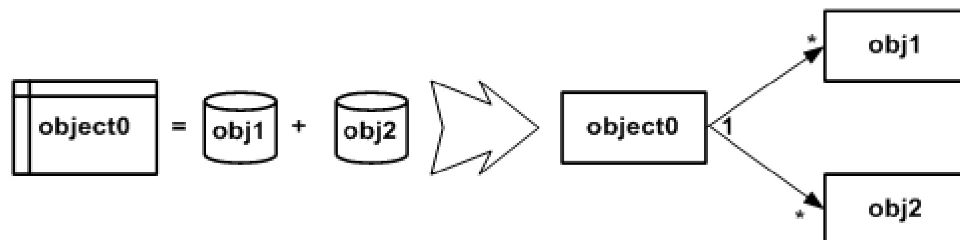


Figure 19 - The Mapping of Composite Data Element

Figure 19 shows a composite data symbol as a result of a combination over two single data objects. In such cases we map both the composite and the single objects to

classes in the target model with navigation links required to build such a model. The corresponding controller operation acts on the composite object but services needed to access the single objects constituents are also created.

Another type of navigation is formed when mapping class attributes whose types do not belong to the set of primitive data types such as *Long* and *String*. In this case, the type is mapped to a class and an association is generated to support the navigation from the main class to that type class. This is illustrated in Figure 20.

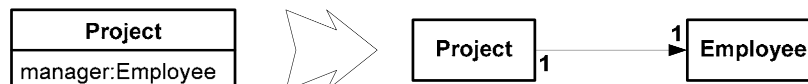


Figure 20 - The Mapping of Non-Primitive Attribute Types

Another type of navigation created with respect to composite objects, occurs when processing tables. Consider a questionnaire, where the User is provided with a list of questions and a set of answers for each. The results are to be saved in a separate table. This is shown in Figure 21. In such cases, the object associated with the operation trigger is created as an aggregation of the objects representing table columns. The resultant data model is depicted in Figure 22.

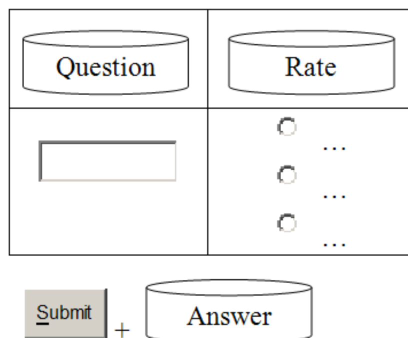


Figure 21 - Composite Objects and Tables

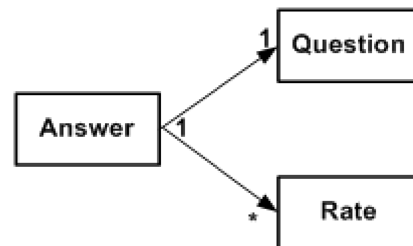


Figure 22 - Data Model created for Figure 21

5.5. Mapping to Specific Platforms: An Example

In this section we will briefly describe a mapping to a sample platform in order to show the feasibility of the method. We use a platform from AndroMDA (2007) resources for implementation and validation purposes. This platform is defined as Java (Programming Language), AndroMDA (Code Generation Framework), MySQL(DBM), DAO (Data Access Mechanism) and Hibernate (2008), ArgoUML (Modeling Tool) and Struts (UI Framework) – better know as *BPM4Struts Cartridge* by the AndroMDA community (AndroMDABPM, 2008), where:

- Hibernate is used as the data access framework. This is required for establishing data transmission amongst different layers. It is based on DAO standards.
- ArgoUML is the modeling tool used for refining the resulting PSM.
- Struts is the UI code framework. Target UI stereotypes and code skeletons are selected from this framework.

Here are some of the definitions regarding this platform:

- A *State Machine Context* is a controller class that is responsible for handling and forwarding operations corresponding to events within a state machine.
- *An action event* is an event that is called when submitting a form in a web page. Action events usually include parameters which are the input fields of the forms.
- *A deferrable event* is an event that invokes a controller operation. Deferrable events are used to assign states with operations.
- *A page parameter* is any output that is either shown or used for output fields on the web page.
- *A value object* is an object that carries information between domain objects and the presentation or data access layer.
- *FrontEndView* is a state stereotype implying that the stereotyped state represents a web page.
- *A signal event* is an event that is usually carried by an incoming transition to a front-end state. A signal event carries output fields to be shown.
- *A call event* is an event on a transition outgoing from a front-end state. A call event carries input fields to be submitted along with a controller operation to be called for performing a required action.

The following rules apply to all models:

1- There must be one controller class per use case

2- There must be a service class per data object

3- Controller classes must be dependent on their objects' service classes

Other specific mapping rules for an APSM model to an AndroMDA-specific PSM are as follows:

4- Every presentation state becomes a state stereotyped as *FrontEndView*

5- For each operation trigger

   a. a signal event is created on the outgoing transition
   b. a deferrable event is created on the next state. If the next state is a choice then the deferrable event is created on the transition ending to the choice.
   c. A controller operation would be generated to be called by the generated deferrable event
   d. The set of input parameters for the signal/deferrable events as well as the controller operation are created based on the UI components belonging to the operation trigger.
   e. For every domain object referred to by a component of an operation trigger an entity domain object and a value object are created. If the domain object requires one of the update, delete, or insert operations then the tag *Manageable* is added to the target object, to force AndroMDA to generate the corresponding operations
      i. There would be a dependency from every entity domain object to the relevant value object
      ii. An operation would be added to the service class to perform the corresponding CRUD operation
      iii. A call would be added to the controller operation to call the service

iv. A dependency would be made from the service class to the domain object so that the service class can access the instances of the domain object

6- Every UI component not owned by a trigger becomes a parameter in the set of parameters on a signal event belonging to the incoming transition.
7- Every choice in the state machine results in the creation of a controller operation that returns a value, based on which the state machine decides which transition to take.

6 - QVT Transformations

The following are general assumptions and conventions upon which the transformations are based:

- The transformations are defined from a Web-based PIM (WPIM) to an Abstract PSM (APSM) but both models are based on the abstract model introduced in Section 4. Below is the description of the contents of two models:
    - WPIM is a subset of the abstract model encompassing the following packages:
        - State Machine
        - UI Structure
        - UI Components
    - APSM has the same set of packages as WPIM plus the followings:
        - Data Model
        - Data Components
- Objects from source are always named with a '1' at the end and those from the target domain are always named with a '2' postfix. This will always remain true even when a relation lacks either a source or target objects.
- States and transitions must be all named.

Code Sample 5 shows the transformation at the topmost level. The transformation is defined by a top relation that transforms an abstract application at the WPIM level to an application belonging to the APSM level.

Code Sample 5 - WPIM to APSM transformation and the top level relation

```
transformation WPIM2APSM (wPIM :WPIM, aPSM : APSM) {
	top relation ApplicationToApplication {
		applicationName:String;

		domain wPIM application1:WPIM::Application {
			title=applicationName,
			stateMachines=stateMachine1:UPIM::StateMachine{}
		}
		domain aPSM application2:APSM::Application {
			title=applicationName,
			stateMachines=stateMachine2:APSM::StateMachine{}
		}
		where {
			StateMachineToStateMachine(stateMachine1, stateMachine2);
		}
	}
}
```

AWPIM state machine is copied to its counterpart from APSM in Code Sample 6.

Code Sample 6 - WPIM State Machine to APSM State Machine QVT Relation

```
relation StateMachineToStateMachine {
	stateMachineName:String;

	domain wPIM stateMachine1:WPIM::StateMachine {
		name=stateMachineName,
		regions=region1:WPIM::Region {
			vertices=vertex1:WPIM::Vertex {}
		}
	}
	domain aPSM stateMachine2:APSM::StateMachine {
		name=stateMachineName,
		regions=region2:APSM::Region {
			vertices=vertex2:WPIM::Vertex {}
		}
	}
	where {
		VertexToVertex(vertex1, vertex2);
	}
}
```

Code Sample 7 shows a relation that copies vertexes from a WPIM to an APSM and then calls another relation to copy those vertexes that are presentation states. We use in-line relations to map outgoing and incoming transitions. The term *in-line relation* refers to implicit mappings that exist between two elements of the same type as subsets of source and target pattern. We usually avoid using in-line relations in order to keep the complexity of our relations low. However, in this case we need to assign the current state as the source and target of the outgoing and incoming transitions respectively. This would have been hard to achieve out of the context of this relation.

Code Sample 7 - WPIM Vertex To APSM Vertex QVT Relation

```
relation VertexToVertex {
	vertexName:String;
	outgoingName:String;
	incomingName:String;

	domain wPIM vertex1:WPIM::Vertex {
		name=vertexName,
		outgoing=outgoingTransition1:WPIM::Transition {
			name=outgoingName,
			source=vertex1,
			target=targetVertex1:WPIM::Vertex{}
		},
		incoming=incomingTransition1:WPIM::Transition {
			name=incomingName,
			target=vertex1,
			source=sourceVertex1:WPIM::Vertex{}
		}
	}
	domain aPSM vertex2:APSM::Vertex {
		name=vertexName,
		outgoing=outgoingTransition1:APSM::Transition {
			name=outgoingName,
			source=vertex2,
```

```
                    target=targetVertex2:APSM::Vertex{}
            },
            incoming=incomingTransition1:APSM::Transition {
                    name=incomingName,
                    target=vertex2,
                    source=sourceVertex2:APSM::Vertex{}
            }
        }
        where {
            PresentationStateToPresentationState(vertex1, vertex2);
        }
}
```

The relation in Code Sample 8 recognizes vertexes that are in fact states and have a presentation and copies them to APSM. The existence of a presentation is automatically checked because it is mentioned as a part of the source and target patterns. In other words, the relation does not start unless the source has a presentation.

Code Sample 8 - WPIM Presentation State To APSM Presentation State QVT Relation

```
        relation PresentationStateToPresentationState {
            domain wPIM vertex1:UPIM::State {
                    presentation=presentation1:UPIM::Presentation {}
            }
            domain aPSM vertex2:APSM::State {
                    presentation=presentation2:APSM::Presentation {}
            }
            where {
                    PresentationToPresentation(presentation1, presentation2);
            }
        }
```

The relation in Code Sample 9 copies presentations from WPIM to APSM and calls another relation to deal with presentations that contain operation triggers.

Code Sample 9 - WPIM Presentation To APSM Presentation QVT Relation

```
        relation PresentationToPresentation {
            presentationName:String;

            domain wPIM presentation1:WPIM::Presentation {
                    name=presentationName,
                    uiComponents=uiComponent1:UPIM::UIComponent{}
            }
            domain aPSM presentation2:APSM::Presentation {
                    name=presentationName,
                    uiComponents=uiComponent2:APSM::UIComponent{}
            }
            Where {
                    OperationTriggerredPresentationToOperationTrigerredPresentation
                    (presentation1, presentation2)
            }
        }
```

The relation in Code Sample 10 finds the operation trigger in the current presentation and copies it.

Code Sample 10 - WPIM Operation Trigger to APSM Operation Trigger QVT Relation

```
relation OperationTriggerredPresentationToOperationTriggerredPresentation {
        domain wPIM presentation1:WPIM::Presentation {
                uiComponents=operationTrigger1:WPIM:OperationTrigger {}
        }
        domain aPSM presentation2:APSM::Presentation {
                uiComponents=operationTrigger2:APSM:OperationTrigger {}
        }
        where {
                OperationTriggerToOperationTrigger(operationTrigger1, operationTrigger2);
        }
}
```

The relation in Code Sample 11 uses the *owningComposite* role that we added to the abstract model, to access the owning presentation of the operation trigger on the target side. The access in needed so that after copying the operation trigger, navigate through the state machine is possible to reach the outgoing transition and the next state for adding further events and parameters. Three calls are embedded in the *where* section:

1. A call to another relation that maps the components of the current operation trigger to required data elements and operations
2. A call to access the owning presentation, through which navigation to the state and access its outgoing transition occurs.
3. A last call to map drop down select components

Code Sample 11 - WPIM Operation Trigger to APSM Operation Trigger QVT Relation

```
relation OperationTriggerToOperationTrigger {
        operationTriggerName:String;
        uiComponentName:String;

        domain wPIM operationTrigger1:WPIM::OperationTrigger {
                name=operationTriggerName,
                uiComponents=uiComponent1:WPIM::UIComponent {name=uiComponentName}
        }
        domain aPSM operationTrigger2:APSM::OperationTrigger {
                name=operationTriggerName,
                uiComponents=uiComponent2:APSM::UIComponent {name=uiComponentName}
                presentation=owningPresentation2:APSM::Presentation{},
        }
        where {
                OperationTriggerQueryToOperationTriggerQuery(operationTrigger1,
                                                        operationTrigger2);
                OperationTriggerToOwningPresentation(operationTrigger1, owningPresentation2);
                SelectToSelect(uiComponent1. UiComponent2);
        }
}
```

The relation in Code Sample 12 defines required data elements and operations for accessing them. Currently, this transformation is limited to performing a filtered select query based on the entries found on the input form. Other possible operations could be eventually added in future. The relation calls two other relations:

1. to create the required domain objects

2. to generate the required parameters that filter the select operation

Code Sample 12 – WPIM Operation Trigger with Query Data to APSM Operation Trigger with Query Data QVT Relation:

```
relation OperationTriggerQueryToOperationTriggerQuery {
    dataName:String;

    domain wPIM operationTrigger1:WPIM::OperationTrigger {
        queryData=queryData1:WPIM::DataElement{name=dataName},
        uiComponents=uiComponent1:WPIM::UIComponent{}
    }
    domain aPSM operationTrigger2:APSM::OperationTrigger {
        queryData=queryData2:APSM::DataElement{
            name=dataName,
            type=class2:APSM::Class {
                name=dataName
            }
            selectOperation=select2:APSM::OperationAdapter {
                name='select'+dataName
            }
        }
    }
    where {
        UIComponentToClass(uiComponent1, class2);
        UIComponentToOperationParameters(uiComponents, selectOperation);
    }
}
```

The relation in Code Sample 13 maps a UI component to a class representing a domain object and calls another relation to map attributes.

Code Sample 13 - WPIM UI Component to APSM Class QVT Relation:

```
relation UIComponentToClass {
    domain wPIM uiComponent1:WPIM::UIComponent {
    }
    domain aPSM class2:APSM::DataElemet {
        ownedAttribute=ownedAttribute2:APSM::Property{}
    }
    where {
        uiComponentToClassAttribute(uiComponent1, ownedAttribute2);
    }
}
```

The relation in Code Sample 14 defines attributes to be assigned to a generated domain object on a target. The *Where* section is to be filled with a set of relations that are required to assign a specific attribute type for every UI component. This is a good illustration of decisions made to guarantee bidirectional mappings. A simple function call such as the one in Code Sample 15 could have been used with the consequence of not being reversible.

Code Sample 14 – WPIM UI Component to APSM Property QVT Relation:

```
relation UIComponentToClassAttribute{
    attributeName:String;
```

```
        domain wPIM uiComponent1:WPIM::UIComponent {
            name=attributeName
        }
        domain aPSM ownedAttribute2:APSM::Property {
            name=attributeName
        }
        where {
            SelectToLong(uiComponent1, ownedAttribute2);
            ...
        }
    }
```

Code Sample 15 – unidirectional WPIM UI Component to APSM Property QVT Relation:

```
    relation UIComponentToClassAttribute{
        attributeName:String;

        domain wPIM uiComponent1:WPIM::UIComponent {
            name=attributeName
        }
        domain aPSM ownedAttribute2:APSM::Property {
            name=attributeName,
            type=getTypeFor(uiComponent1)
        }
    }
```

The relation in Code Sample 15 is not reversible because the target domain is firmly dependent to the source domain. Running it from target to source is impossible.

Now, lets go back to Code Sample 11 and follow the path taken by the second relation call in the *where* section. This relation is shown in Code Sample 16. The relation only takes the transformation one step closer to the goals of copying the components of an operation trigger to an outgoing transition, by finding the owning presentation of the current operation trigger and calling another relation to find the state that owns this presentation.

Code Sample 16 - WPIM Operation Trigger to APSM Presentation Owning Operation Trigger QVT Relation

```
    relation OperationTriggerToOwningPresentation {
        domain wPIM operationTrigger1:WPIM::OperationTrigger {
        }
        domain aPSM owningPresentation:APSM::Presentation {
            state=owningState2:APSM::State{}
        }
        where {
            OperationTriggerToOwningState(operationTrigger1, owningState2);
        }
    }
```

The relation in Code Sample 17 finds the state owning a presentation and forwards the source operation trigger to the outgoing transition for event assigning purposes.

Code Sample 17 - WPIM Operation Trigger to APSM State Owning Presentation

```
relation OperationTriggerToOwningState {
```

```
	domain wPIM operationTrigger1:WPIM::OperationTrigger {
	}
	domain aPSM owningState2:APSM::State {
		outgoing=outgoingTransition2:APSM::Transition{}
	}
	where {
		OperationTriggerToOutgoingTransition(operationTrigger1, outgoingTransition2);
	}
}
```

Finally, the relation in Code Sample 18 finds an appropriate transition on which to create an event. The whole path taken is another example of disposition needed to make the transformation bidirectional. Alternatively, one could rewrite Code Sample 11 as Code Sample 19. The relation in Code Sample 19 is however, not reversible as it loses track of owning elements when trying to create the event on the outgoing transition.

Code Sample 18 - WPIM Operation Trigger To APSM Outgoing Transition

```
	relation OperationTriggerToOutgoingTransition {
		domain wPIM operationTrigger1:WPIM::OperationTrigger {
		}
		domain aPSM outgoingTransition2:APSM::Transition {
			events=event2:APSM::Event{}
		}
		where {
			OperationTriggerToTransitionEvent(operationTrigger1, event2);
		}
	}
```

Code Sample 19 – Unidirectional WPIM Operation Trigger to APSM Operation Trigger QVT Relation

```
relation OperationTriggerToOperationTrigger {
	operationTriggerName:String;

	domain wPIM operationTrigger1:WPIM::OperationTrigger {
		name=operationTriggerName
	}
	domain aPSM operationTrigger2:APSM::OperationTrigger {
		name=operationTriggerName
	}
	where {
		OperationTriggerQueryToOperationTriggerQuery(operationTrigger1,
								operationTrigger2);
		outgoingTransition=operationTrigger2.owningPresentation.owningState.outgoing;
		OperationTriggerToOutgoingTransition(operationTrigger1, outgoingTransition);
	}
}
```

The relation in Code Sample 20 takes care of the creation of an event required on an outgoing transition. The event name is copied form the operation trigger. Another relation is called to assign the parameters to the event according to the set of UI components as discussed in Section 4.

Code Sample 20 - WPIM Operation Trigger To APSM Transition Event

```
	relation OperationTriggerToTransitionEvent {
		eventName:String;
```

```
		domain wPIM operationTrigger1:WPIM::OperationTrigger {
			name=eventName,
			uiComponents=uiComponents1:WPIM::UIComponent{}
		}
		domain aPSM event2:APSM::Event {
			name=eventName,
			parameters=parameter2:APSM::Parameter{}
		}
		where {
			UIComponentToParameter(uiComponents1, parameter2);
		}
	}
```

The relation in Code Sample 21 maps source UI components to event parameters using the same mechanism as in Code Sample 10.

Code Sample 21 - WPIM UI Component To APSM Event Parameter

```
	relation UIComponentToParameter {
		parameterName:String;

		domain wPIM uiComponent1:WPIM::UIComponent {
			name=parametyerName
		}
		domain aPSM parameter2:APSM::Parameter {
			name=parameterName
		}
		where {
			SelectToLong(uiComponent1, parameter2);
			...
		}
	}
```

The relation in Code Sample 22 maps a select component to another select component. The relation calls two more relations to:

- create the domain object required to model the information associated with the select components
- create the operation required to load all the instance objects

Code Sample 22 - UPIM UI Select Component To APSM UI Select Component

```
	relation SelectToSelect {
		selectName:String;

		domain wPIM select1:WPIM::Select {
			name=selectName,
			dataSource=dataSource1:WPIM::DataComposite{}
		}
		domain aPSM select2:APSM::Select {
			name=selectName,
			dataSource=dataSource2:APSM:DataComposite{}
		}
		where {
			DataSourceToDataSource(dataSource1, dataSource2);
			SelectOperationToSelectOperation(dataSource1, dataSource2);
		}
```

```
}
```

The relation in Code Sample 23 maps the data composite element that supplies the select component.

Code Sample 23 - WPIM Data Source To APSM Data Source

```
relation DataSourceToDataSource {
	dataSourceName:String;

	domain wPIM dataSource1:WPIM::DataComposite {
		name=dataSourceName,
		type=type1:WPIM::Class{}
	}
	domain aPSM dataSource2:APSM:: DataComposite {
		name=dataSourceName,
		type=type2:APSM:Class{}
	}
}
```

The relation in Code Sample 24 creates a select operation required to supply a select component. Unlike the select operation in Code Sample 11, this operation does not require any parameter as it loads all the instance objects.

Code Sample 24 - WPIM Select Operation To APSM Select Operation

```
	relation SelectOperationToSelectOperation {
		domain wPIM dataSource1:WPIM::DataSource {
		}
		domain aPSM dateSource2:APSM::DataSource {
			selectOperation=selectOperation2:APSM::OperationAdapter{}
		}
		where {
			SelectOperationToOperationAdapter(dataSource1, selectOperation2);
		}
}
```

Section 7- A Case Study

Different case studies are being considered in order to validate our approach. In this section, we will discuss one of these case studies; an Election Management System (EMS) introduced by Lethbridge and Laganière (2001). The intent of this system is to manage the information regarding elections and polls. The system provides facilities for data-related activities and management of the voting process. It also provides some information to journalists.

Figure 23 presents the use case diagram of EMS. A full implementation of EMS would encompass more use cases. For example, we have not included the use cases required for deleting or updating information elements. We have also simplified the definition by making some assumptions such as assigning only one poll to each polling station.

Our implementation concerns a subset of the EMS consisting in use cases:

1- Login
2- Define Election

3- Define Seat
4- Define Poll
5- Open Poll
6- Vote
7- Close Poll

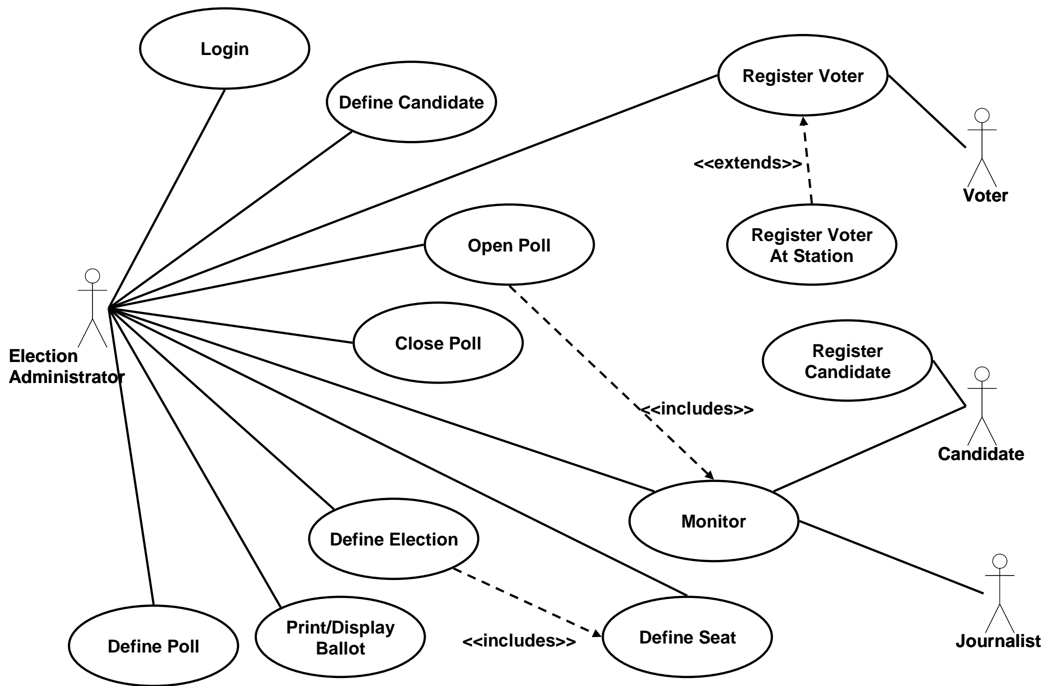


Figure 23 – EMS Use-Case Diagram

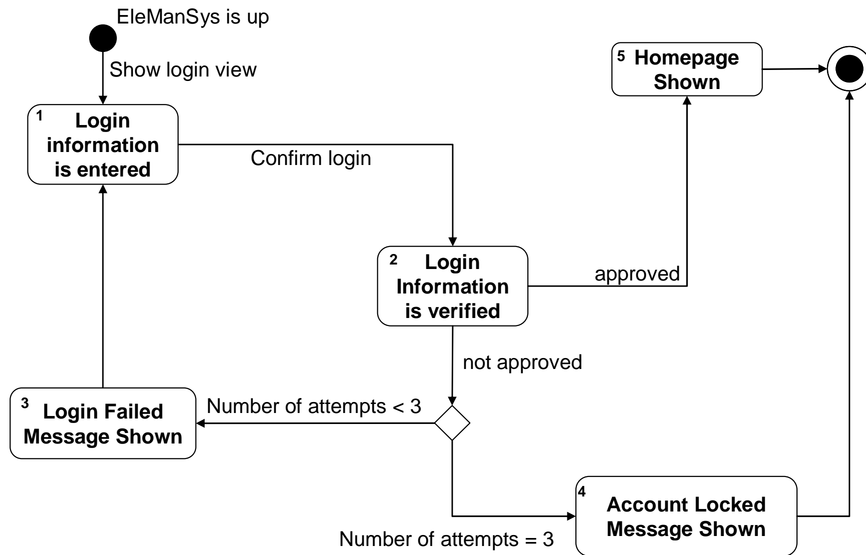


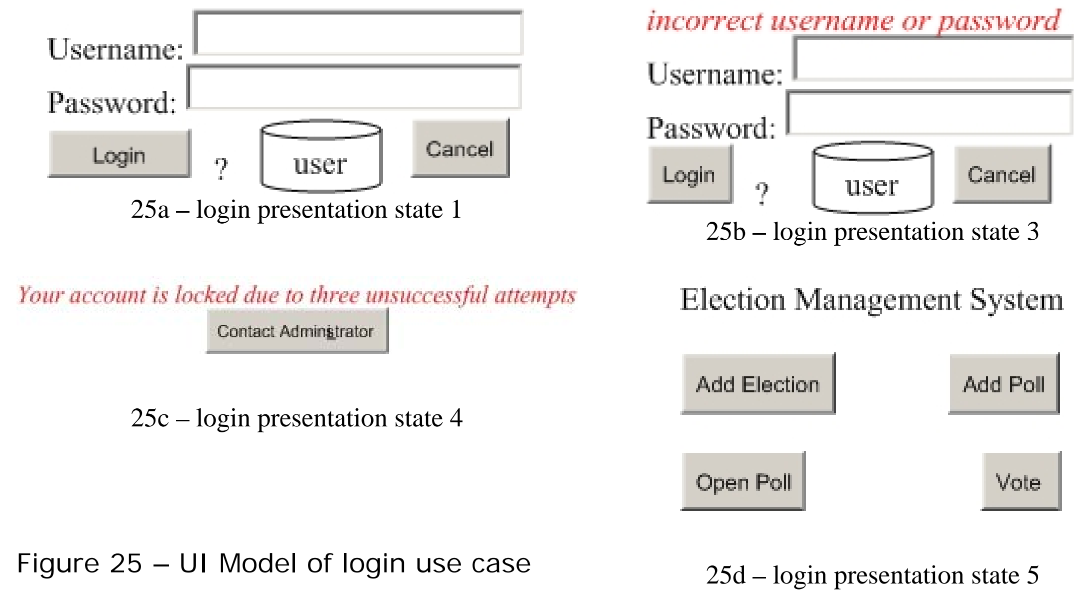
Figure 24 – log-in state machine

7.1. Administrator Login Use Case

Figure 24 shows the state machine describing the behavior of use case Login. The administrator attempts to login. The machine verifies the login information and shows a

homepage when the login information is valid. If the information is invalid and the administrator has not reached a maximum of three attempts, he/she is be given another chance to login. Otherwise, the account will be locked.

Figure 25 presents the UI model of the login state machine in four different sections for states 1, 3, 4, and 5. The proper mechanism of login/logout use cases is different than just a simple query checking mechanism and depends on the target platform and selected security technology. We abide here by an abstract mechanism.



25a – login presentation state 1

25b – login presentation state 3

25c – login presentation state 4

25d – login presentation state 5

Figure 25 – UI Model of login use case

We now traverse the QVT relations over the login state machine to see the results. The transformation happens as follows:

- The relations in Codes 5 to 11 simply generate the application, state machine, states, transitions, presentations and UI components as they appear in the source model.
- The relation in Code Sample 12 results in the generation of a class named *User* and a select operation called *selectUser*.
- The relations in Code Sample 13 and 14 add two attributes to the class, *User*:
    - username:String
    - password:String
- The relations in Codes 16 to 18 traverse the model to reach the outgoing transitions of every presentation state.
- The relations in Codes 20-21 results in the generation of the following events:
    - outgoing state 1 event, *login(username:String, password:String)*
    - outgoing state 3 event, *login(username:String, password:String)*
    - outgoing state 4 event, *contactAdministrator*
    - outgoing state 5 events, *addElection, addPoll, openPoll, vote*,
- Relations to generate the controller operation has not been formally defined but the mapping rules result in the generation of two controller operations:
    - isLoginApproved:boolean created on transition out of state 2
        - Added Remarks
            - call selectUser(username, password) service

                - return true if (username, password) was found in User
    - isAttemptsLimitReached
        - Added Remarks
            - return true if isAttemptsLimitReached

7.2. Define Election Use Case

We move on to the next use case in our outlined scenario, which is when the administrator defines and adds a new election. The behavior of this use case is described as the state machine in Figure 26.

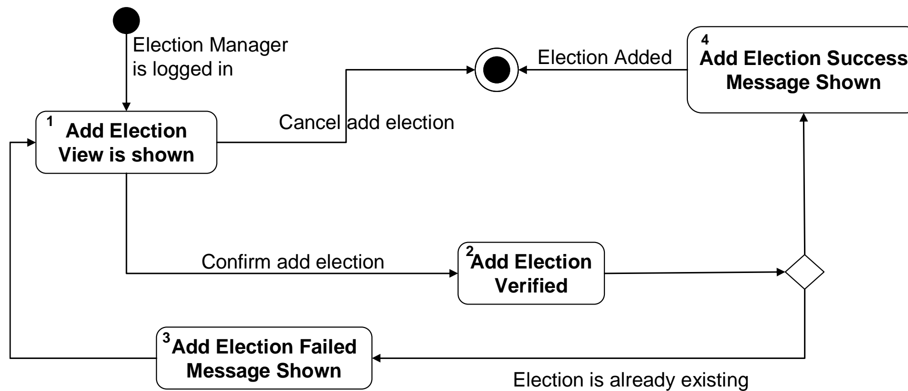


Figure 26 - Add Election state machine

As Figure 26 shows, the User views the Add Election page. The User can choose to cancel, which ends the use case. After election information is entered and confirmed, the EMS verifies it; another trial is initiated if the same election is found in the database. When the addition of an election is successful, a message is shown and the use case returns to the homepage.

Figure 27 contains the presentation attached to states 1 and 3 in Figure 26.

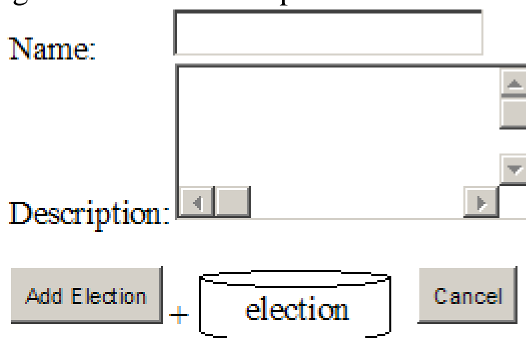


Figure 27a – Add Election presentation state 1

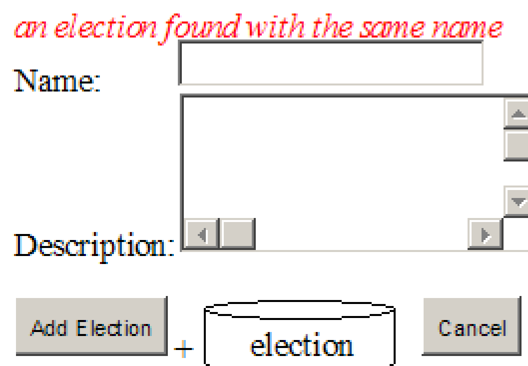


Figure 27b – Add Election presentation state 3

Figure 27 – Add Election presentation states

We now apply the QVT relations to the Add Election state machine:

- The relations in Codes 5 to 11 simply generate the application, state machine, states, transitions, presentations and UI components as they appear in the source model.

- The relation in Code Sample 12 results in the generation of a class named *Election* and an insert operation called *insertElection*.
- The relations in Codes 13 and 14 add two attributes to the class, *Election*:
  - name:String
  - description:String
- The relations in Codes 16 to 18 traverse the model to reach the outgoing transitions of every presentation state.
- The relations in Codes 20-21 result in the generation of the following events:
  - outgoing state 1 event, *addElection(name:String, description:String)*
  - outgoing state 3 event, *addElection(name:String, description:String)*
- Relations to generate the controller operation has not been formally defined but the mapping rules result in the generation of two controller operations:
  - isElectionInserted:boolean created on transition out of state 2
    - Added Remarks
      - call insertElection(name, description) service
      - return false if (name, description) was found in Election; return true if the insertion was successful

7.3. Add Seat Use Case

Figure 28 shows the state machine describing the behavior of Use Case Add Seat, which defines a position to be competed upon in an election. Figure 29 contains the presentation regarding states 1 and 3. This use case is adds a seat to an existing election. The User must select an election and specify information including the maximum number of incumbents for the seat. This use case illustrates the usage of a select box.

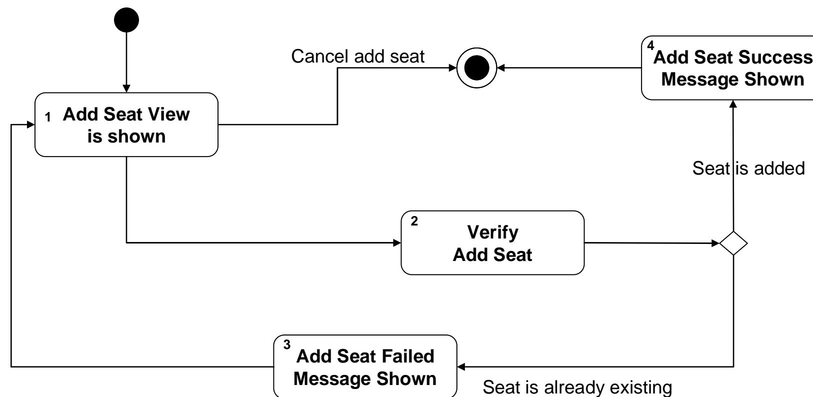
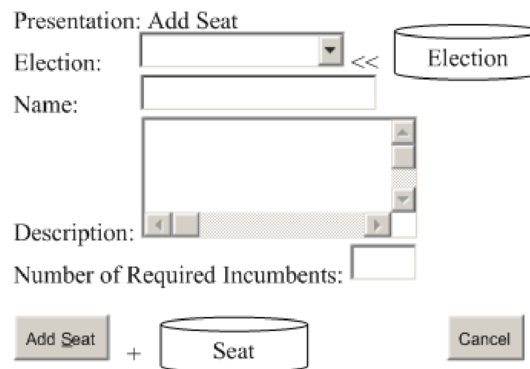


Figure 28 - Add Seat state machine

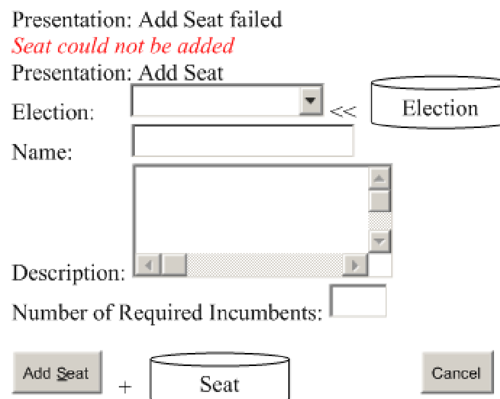
We now apply the QVT relations to the Add Seat state machine:

- The relations in Codes 5 to 11 simply generate the application, state machine, states, transitions, presentations and UI components as they appear in the source model.
- The relation in Code Sample 12 result in the generation of a class named *Seat* and an insert operation called *insertSeat*.
- The relations in Codes 13 and 14 add two attributes to the class *Seat*:
  - name:String
  - description:String
  - election:Long
  - incumbents:Integer

- The relations in Codes 16 to 18 traverse the model to reach the outgoing transitions of every presentation state.
- The relations in Codes 20-21 result in the generation of the following events:
  - outgoing state 1 event, *addSeat(election:Long, name:String, description:String, incumbents:Integer)*
  - outgoing state 3 event, *addSeat(election:Long, name:String, description:String, incumbents:Integer)*
- The relations in Codes 22-24 result in the generation of a select operation called *selectElections* that loads all the instances from the *Election* objects.
- Relations to generate the controller operation has not been formally defined but the mapping rules result in the generation of one controller operation:
  - isSeatInserted:boolean created on transition out of state 2
    - Added Remarks
      - call insertSeat(election, name, description, incumbents) service
      - return false if (election, name, description, incumbents) was found in Seat; return true if the insertion was successful
- An event is added to the incoming transition that calls the controller operation to load all the elections using the selectElections service.



29a – Add Seat presentation state 1



29b – Add Seat presentation state 3

Figure 29 – Add Seat presentation states

7.4. Add Poll Use Case

Once an election is defined, the User needs to assign a Poll to that election to make the voting process possible. The behavior of the Add Poll use case is shown in Figure 30. Figure 31 shows the presentation of the state 1 of the Add Poll state machine. A poll is added by selecting an election to assign the poll to. The poll has a location, date and time.

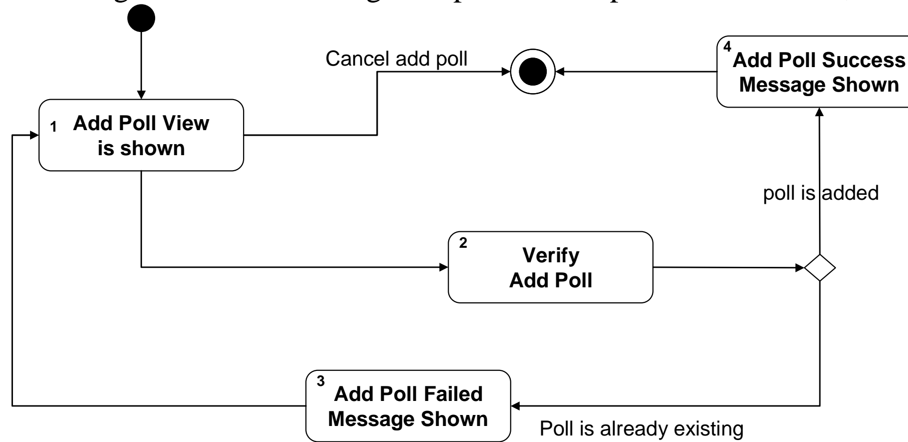


Figure 30 - Add Poll state machine

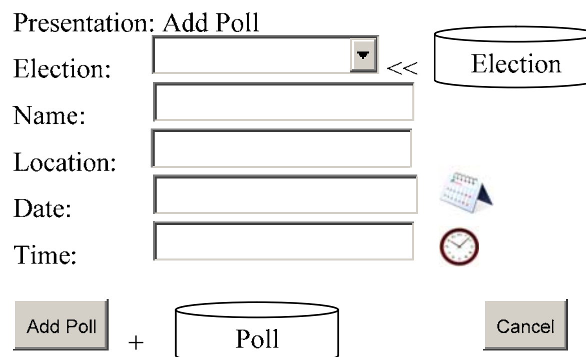


Figure 31 - The Add Poll Presentation

We now apply the QVT relations to the Add Poll state machine:

- The relations in Codes 5 to 11 simply generate the application, state machine, states, transitions, presentations and UI components as they appear in the source model.
- The relation in Code Sample 12 result in the generation of a class named *Poll* and an insert operation called *insertPoll*.
- The relations in Codes 13 and 14 add two attributes to the class, *Poll*:
    - name:String
    - location:String
    - election:Long
    - date:Date
    - time:Time
- The relations in Codes 16 to 18 traverse the model to reach the outgoing transitions of every presentation state.
- The relations in Codes 20-21 result in the generation of the following events:

- outgoing state 1 event, *addPoll(election:Long, name:String, location:String, incumbents:Integer)*
- outgoing state 3 event, *addSeat(election:Long, name:String, description:String, incumbents:Integer)*

- The relations in Codes 22-24 result in the generation of a select operation called *selectElections* that loads all the instances from the *Election* objects.
- Relations to generate the controller operation has not been formally defined but the mapping rules result in the generation of one controller operations:
  - isPollInserted:boolean created on transition out of state 2
    - Added Remarks
      - call insertPoll(election, name, location, incumbents) service
      - return false if (election, name, location, incumbents) was found in Poll; return true if the insertion was successful
  - An event will be added to the incoming transition that calls the controller operation to load all the elections using the *selectElections* service. This in turn results in the creation of an object representing a collection of *Election* objects.

7.5. Open Poll Use Case

Prior to voting, one needs to open a poll. This use case opens the poll so that the voting terminals become dedicated to that poll voting process.

Figure 32 presents that the User must first select an election in order to populate the list of polls. The presentation of the state, *Open Poll View is shown* is depicted in Figure 33. Compared to previous use cases, this is a new situation where the controller is called before the termination of the use case. Therefore, we will need more than one controller operations. The use case performs three controller operations: two that are used to load elements from the election and poll data sources; and a third one to update the Poll object in order to change its status to *Open*. The star placed on the Election select component denotes the default event holder of the presentation. The same presentation model is used for the *Populate Polls List* state without the star, which means that the *Open Poll* submit button is used as the default trigger event in that case.

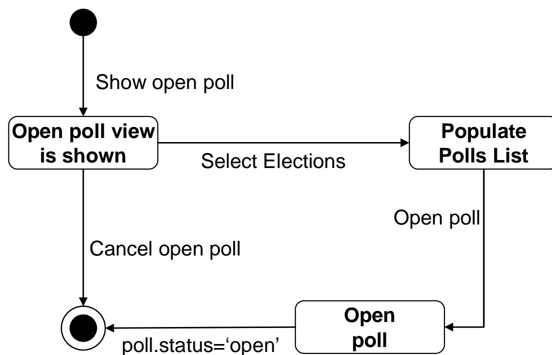


Figure 32 - The Open Poll State Machine

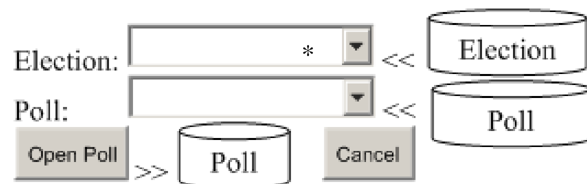


Figure 33 - Presentation of Open Poll State Machine

We now apply the QVT relations to the Add Poll state machine:

- The relations in Codes 5 to 11 simply generate the application, state machine, states, transitions, presentations and UI components as they appear in the source model.

- The relations in Code Sample 12 result in the generation of the operation *updatePoll*.
- The relations in Codes 16 to 18 traverse the model to reach the outgoing transitions of every presentation state.
- The relations in Codes 20-21 result in the generation of the following events:
  - outgoing state 1 event, *loadPoll(election:Long)*
  - outgoing state 3 event, *updatePoll(poll:Long)*
- The relations in Codes 22-24 result in the generation of two select operations called *selectElections* that loads all the instances from the *Election* objects and *selectPolls,* which loads the corresponding poll objects.
- Relations to generate the controller operation has not been formally defined but the mapping rules result in the generation of one controller operations:
  - isPollUpdated:boolean created on transition out of state 3
    - Added Remarks
      - call updatePoll(election) service
      - return false if Poll was not updated
  - An event is added to the incoming transition of state 1 that calls the controller operation to load all the polls using the *selectElections* service.
  - An event is added to the incoming transition of state 2 that calls the controller operation to load all the polls using the *selectPolls* service. The latter results in the generation of a collection of *Poll* objects.

7.6. Vote Use Case

There could be several different scenarios for voting depending on the size and level of an election. In this case, we assume that voters login to a terminal; the system provides a default ballot, in which the voter can see the positions and the names of candidates. The voter fills the ballot out and confirms his/her vote. It is assumed that only one poll at a time can be opened in a location. A voter can only login once while a poll is open.

The state machine in Figure 34 shows the voting use case. The state machine has a few presentation states. Since we already covered a login use case, we focus on the state, *Ballot Shown*. In this state, a presentation containing a list of possible seats and candidates are shown. Figure 35 shows this presentation. The ballot is presented using a two-columns table. The first column lists the possible seats. The second column shows the list of candidates available per seat.

- The relation in Code Sample 12 results in the generation of the operation *addBallot*.
- The relations in Codes 20-21 result in the generation of the following event:
  - outgoing state 1 event, *addBallot(seat:Long, candidates:Long[])*
- The relations in Code Sample 22-24 result in the generation of two select operations called *selectSeats* that loads all the instances from the *Seat* objects that belong to the currently open poll and *selectCandidates,* which loads the corresponding *Candidate* objects that compete for that specific seat. This mechanism has not been formally defined in the mappings.

The voting use case also includes a special case of composite objects shown in a table. In this case, a domain object, *Ballot* is generated with two attributes/associations:

- *seat:Seat*
- *candidates:Candidate[]*

7.7. Close Poll Use Case

The last use case in this case study calculates and reports the final results of a poll. This is shown in Figure 36 with the corresponding presentation in Figure 37.

The table in Figure 37 is composed of a view created based on entries from Ballot objects. The application should read all the ballots and calculates the most assigned vote to every candidate.

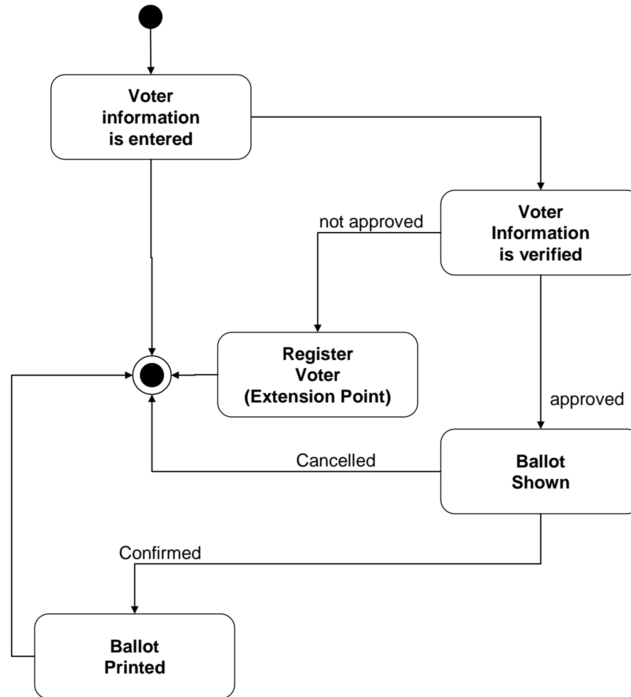


Figure 34 - Voting State Machine

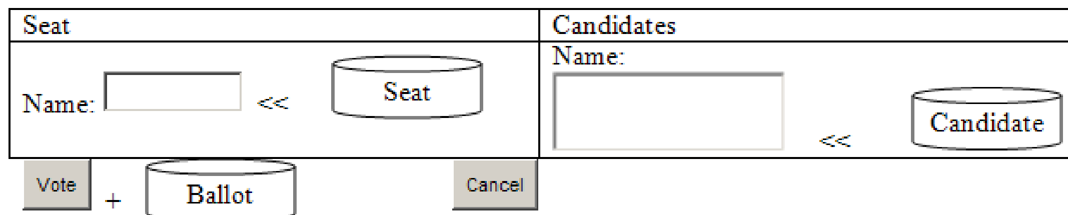


Figure 35 - Presentation of a Ballot for Voting purpose

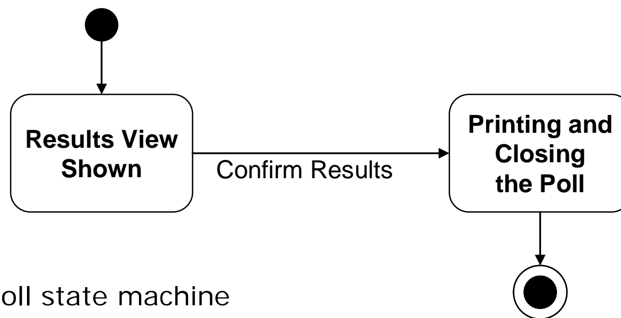


Figure 36 - Close Poll state machine

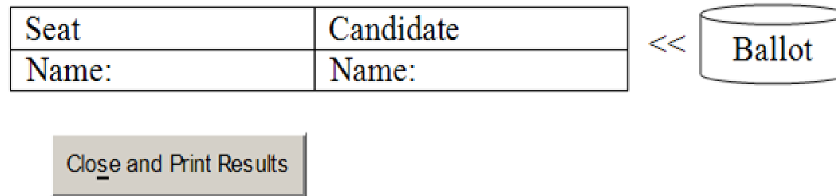


Figure 37 - Close Poll Presentation

7.8. The Generated Application

In this section, we describe the generated abstract web application by summarizing the models generated so far. The structure of the application is understood in terms of the presentation states and the way they call or encompass each other. The call structure of the created EMS for a successful scenario is shown in Figure 38.

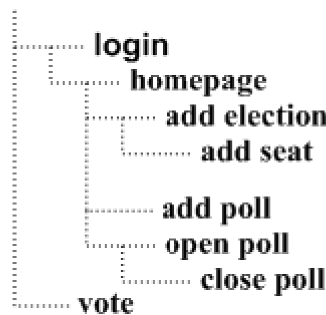


Figure 38 - The structure of EMS in a happy scenario

Figure 39 shows a simplified version of the class model of EMS. Classes that are not generated as the result of applying the mappings to the use cases covered in this section are shadowed. One should note that although the domain model should belong to the PIM, our method extracts domain classes automatically. There are also other objects and associations the mappings create in order to manage collections of *Poll*s and *Election*s, which are not covered Figure 39.

Finally, Table 3 shows the controller operations created per use case and the usage of data services projected in these controllers. The state machines and presentation models are copies of the PIM with data associations removed. Therefore, we have not inserted them in this section.

7.9. Backward Change

As stressed earlier, one of our goals is the ability to support backward changes from the APSM to the PIM. In this section, we review an example illustrating such changes. Consider the Open Poll use case. Suppose a scenario, in which the developer finds out that the system should proceed with monitoring the poll process after the poll opens. This functionality was not considered in Section 7.5. Suppose the developer adds an event on the transition going out of state Open Poll in the APSM version of Figure 35 with an event called monitor(poll:Long) in order to support the new functionality.

First we need to read the relevant relations in reverse order. Code Sample 28 contains the relevant relation read in reveres direction.

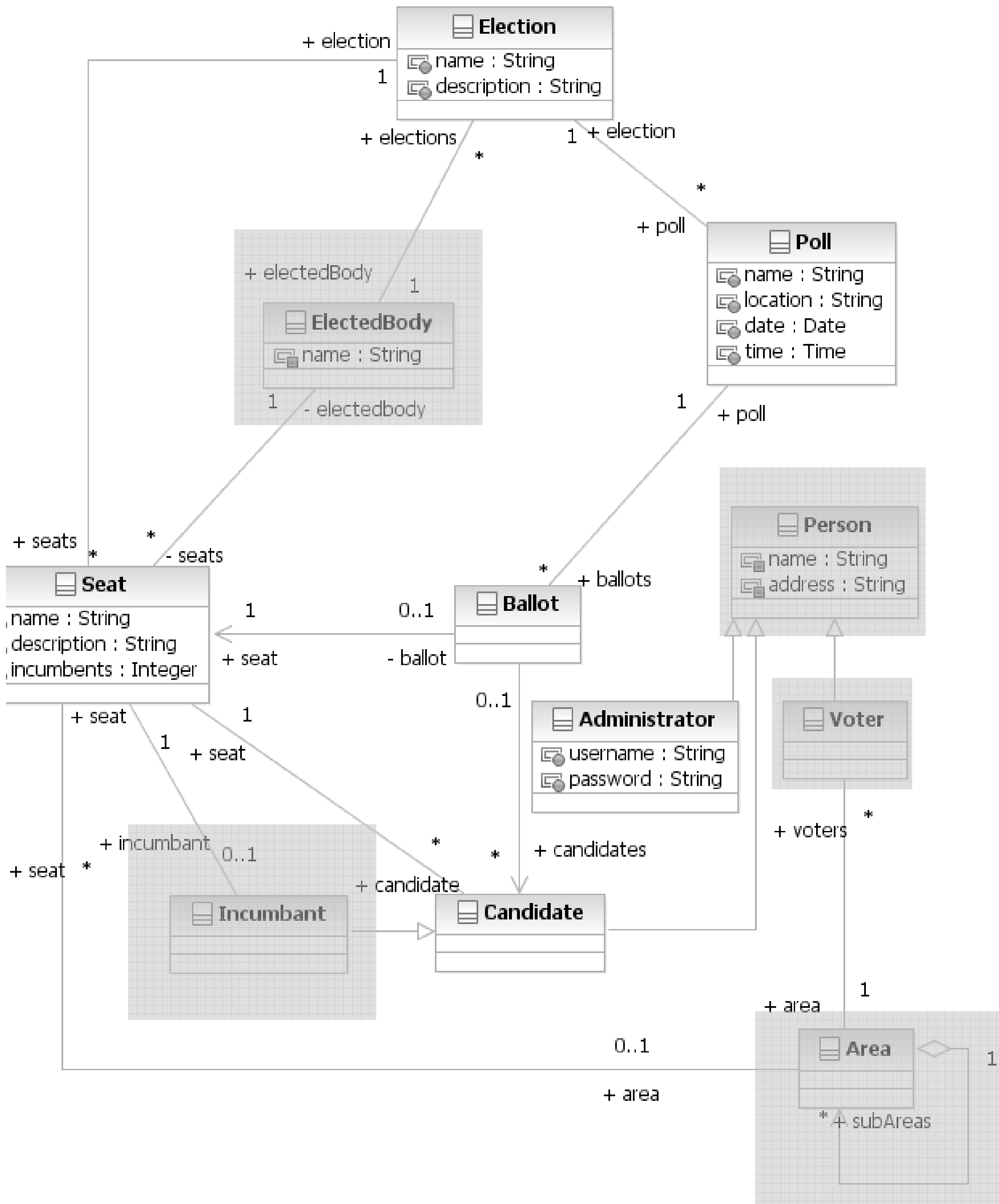


Figure 39 - The class model of EMS according to use cases covered in this section

Code Sample 28 - APSM Transition Event To WPIM Operation Trigger QVT Relation (Reverse of Code Sample 20)

```
relation OperationTriggerToTransitionEvent {
        eventName:String;

        domain aPSM event2:APSM::Event {
```

```
                    name=eventName,
                    parameters=parameter2:APSM::Parameter{}
            }
            domain wPIM operationTrigger1:WPIM::OperationTrigger {
                    name=eventName,
                    uiComponents=uiComponents1:WPIM::UIComponent{}
            }
            where {
                    UIComponentToParameter(uiComponents1, parameter2);
            }
    }
```

The relation in Code Sample 28, results in the generation of an operation trigger named *monitor* in the PIM, and the invocation of another relation to map UI components of the operation trigger to event parameters. This relation should be read in reverse as shown in Code Sample 29.

Code Sample 29 - APSM Event Parameter To WPIM UI Component QVT Relation (Reverse of Code Sample 21)

```
    relation UIComponentToParameter {
            parameterName:String;

            domain aPSM parameter2:APSM::Parameter {
                    name=parameterName
            }
            domain wPIM uiComponent1:WPIM::UIComponent {
                    name=parametyerName
            }
            where {
                    SelectToLong(uiComponent1, parameter2);
                    ...
            }
    }
```

The relation in Code Sample 29 creates a UI component named *poll* in the APSM version of Figure 35. The type of this component will then be defined by calling the relations in *Where* section, which in this case maps parameter *poll* to a drop-down select box. The administrator could then select a poll from the list to monitor. Note that even if there is only one election in process, there could be more than one poll in progress in different sites.

7.10. Mapping to AndroMDA-specific Platform

We have developed the EMS application using the AndroMDA-specific rules in Section 5.5. In this section, we verify the application of the AndroMDA-specific rules to a few use cases to describe the results of the mappings.

Before going through the example, we will review the mechanism to create controller and service operations. According to the mapping rule 1, there must be one controller per each use case. All the controller operations will be assigned to this class. Controller operations are generated for two different reasons:

1- Performing a CRUD operation: in this case a call to the corresponding data access service operation will be also inserted within the operation. For each CRUD operation, there would be only one controller operation.
2- Determining the outgoing transition of a choice: the state machine will simply call the controller operation to decide which path to continue on.

The following are the elements created in the AndroMDA-specific model with respect to the Login use case:

- The service class, *userService* with operation (Mapping Rule 2)
  - *selectUser(username, password):User (Mapping Rule 5.e.ii)*
- The controller class, *loginController* (Mapping Rule 1) with operations
  - *is LoginApproved(username, password):boolean (Mapping Rule 5.c and 7)*
    - calls the selectUser service (Mapping Rule 5.e.iii)
  - *isAttemptsLimitReacged():boolean (Mapping Rule 7)*
- A dependency from *loginController* to *userService (Mapping Rule 3)*
- A domain object, 'user' with the attributes, *username* and *password* both typed as *String*. The object will be stereotyped as *Entity*. (Mapping Rule 5.e)
- A value object, *UserVO* that carries the same information amongst different layers of application. (Mapping Rule 5.e)
- A *login* signal event on the transition going out of state 1 in Figure 27 (Mapping Rule 5.a) carrying the same parameters as the attributes of *User (Mapping Rule 5.b).*
  - *username:String*
  - *password:String*
    - Password parameter will be tagged by AndroMDA-specific tags to denote that this is presented as a password input box, which is differentiated from regular input boxes.
- A dependency from *User* to *UserVO (Mapping Rule 5.e.i)*

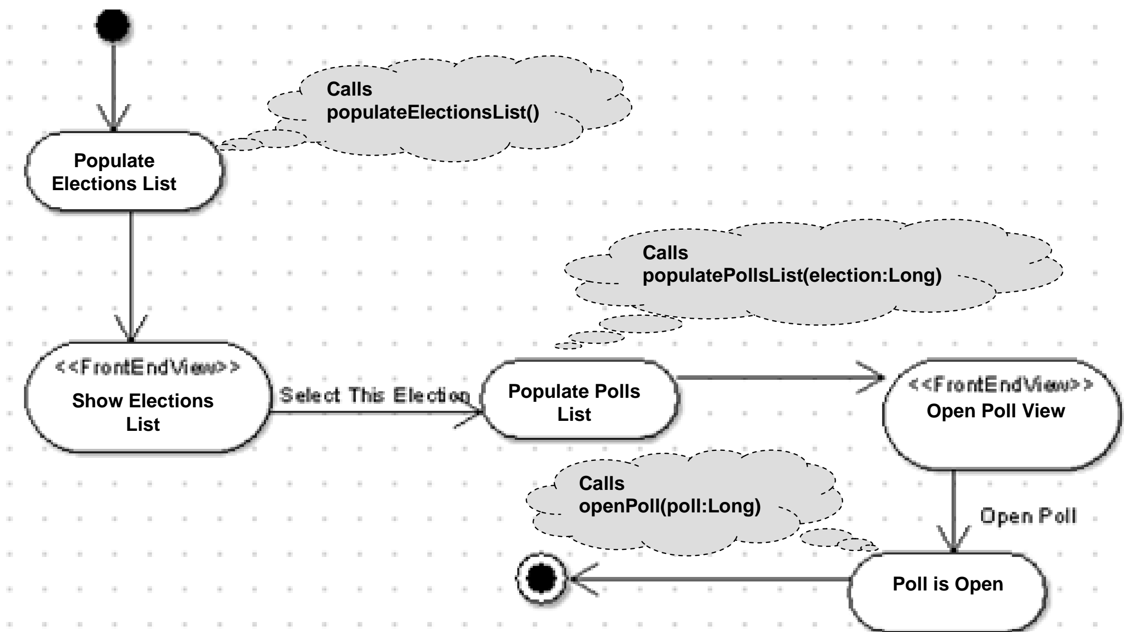


Figure 40 – Open Poll state machine in AndroMDA-specific platform

The Open Poll use case is a case in which it is not possible to create the exact UI model and state machine in the AndroMDA-specific side. The reason is that in the current version of AndroMDA, one cannot automatically build the select events as required by the model of Figure 33 (See AndroMDA_Forum, 2008). As a result one has to break the UI model into two different sections: one that selects the election and the other that selects the poll to open. Figure 40 shows the resultant state machine in AndroMDA side for a simplified Open Poll use case.

The following list shows the elements generated in corresponding to the *Open Poll* use case starting from *Populate Polls List* state:

- The service class, *pollService* with operation (Mapping Rule 2)
  - *updatePoll(poll:Long):boolean (Mapping Rule 5.e.ii)*
  - *selectPolls(election:Long):Poll[](Mapping Rule 5.e.ii)*
- The controller class, *openPollController* (Mapping Rule 1) with operations
  - *populatePollsList()(Mapping Rule 5.c and 7)*
    - calls the *selectPolls* service (Mapping Rule 5.e.iii)
  - *isAttemptsLimitReacged():booleanopenPoll()(Mapping Rule 5.c and 7)*
    - calls the *updatePoll* service (Mapping Rule 5.e.iii)
- A dependency from *openPollController* to *pollService (Mapping Rule 3)*
- The domain object, ‘poll’ will be stereotyped as *Manageable*. *(Mapping Rule 5.e)*
- A value object, *PollVO* that carries the same information amongst different layers of application. (Mapping Rule 5.e)
- Another value object *PollVO[]* to carry the collection of polls returned by *selectPolls* service. (Mapping Rule 5.e)
- A *populatePollsLost* signal event on the transition going out of the first state of Figure 34 (Mapping Rule 5.a) carrying the ID of the selected election as its parameter *(Mapping Rule 5.b)*.
- An *openPoll* signal event on the transition going out of State *Open Poll* in Figure 34 (Mapping Rule 5.a) carrying the same parameters as the attributes of *Poll*.
- Dependencies from *Poll* to *PollVO (Mapping Rule 5.b)*

The *Open Poll* use case has an interesting feature, which is the transmission of the selected election parameter to the operations required to populate the polls list. This is necessary because, one needs to select an election prior to selecting a poll. This is simply done by using the same parameter name for both the selection trigger fired from the elections list and the controller operation used for populating the polls list.

In case of the voting use case, the rules must result in an object that will be used for supplying data to the table. This is basically a composite data object, which is supplied by all the candidates running for a seat. In this case an intermediate object which is composed of both Seat and Candidate objects will be generated. This has not been formally defined in the AndroMDA-specific mappings yet but has been automatically done for the current implementation.

We return to our technique to assign a controller to each use case and a service to each entity. Therefore, a controller class could be dependent on more than one service. For example, in case of voting use case different entities are required; these are *Candidate, Seat, Poll* and *Ballot*. Thus the following dependencies will be generated:

1- Controller to Poll; in order to access the information regarding a specific poll.

2- Controller to Seat: in order to load all the seats candidates compete for in a certain poll
3- Controller to Candidate: used for retrieving the list of candidates competing for a specific seat
4- Controller to Ballot: in order to generate a ballot composed of the information regarding the seats and candidates.

Further, the pseudo-code automatically generated during the PIM-to-APSM transformation could be automatically transformed to the actual code used for building controller and service operations. For example, one can see that in several occasions, we need to populate a drop-down list of data items. Code sample 30 shows the typical code fragment used for controlling such elements, in which the term *Element* will be replaced by the exact data element in each case, e.g. *Election* in Figure 34.

Code Sample 30 – Automatically-generated code for populating select boxes

```
ElementVO[] elements = getElementService().selectAllElements();
List elementList = new ArrayList(Arrays.asList(elements));

form.setElementBackingList(elementList, "id", "elementName");
```

Code Sample 30 invokes a service operation, for which Code Sample 31 provides a template:

Code Sample 31 – Automatically generated code for selectAllElements service

```
Collection elementVOs = getElementDao().loadAll(ElementDao.TRANSFORM_ELEMENTVO);
 return (ElementVO[])elementVOs.toArray(new ElementVO[0]);
```

Code Samples 30-31 could be seen as the results of transforming Code Sample 4.

Table 3 - Use Cases, Controller Operations and Data Services of the EMS

Use Case	Controller Operation	Data Service
Login	isLoginApproved	selectUser
	isAttemptsLimitReached	none
Add Election	isElectionInserted	insertElection
Add Seat	isSeatInserted	insertSeat
Add Poll	populateElection	selectElections
	isPollInserted	insertPoll
Open Poll	populatePoll	selectPolls
	populateElection	selectElections
	isPollUpdated	updatePoll
Vote	isBallotInserted	insertBallot
Close Poll	isPollClosed	updatePoll

8. Conclusion

We described our proposal for an automated method for the generation of an abstract PSM from a PIM in the context of web-based information systems. The PIM is defined in terms of use cases and their presentation models. The behavior of use cases is provided as state machines. The method automatically generates domain objects and the APSM. The APSM could be later mapped to specific PSMs that are appropriate for certain web development and hosting platforms. We use QVT relations to model bi-directional mappings. Therefore, our approach supports the generation of an APSM from a PIM as well as reflecting required changes from an APSM to a PIM.

A number of customizations were required to adapt the abstract model to our approach. Refinements made include the addition of several navigation links to enable the access to owning objects, and to associate data elements and data sources with operation triggers for the automated generation of data services. The following list shows how our model covers different aspects of a web application:

- The hypertext content is supported by the data-centric UI components.
- The navigation is modeled using triggers that could be operation or hyperlink triggers.
- The behavior is supported by the usage of state machines related to the UI model.
- The presentation is modeled by the UI structure and components.
- The data access is supplied by operation adapters.

Future work is mainly devoted to more experiment with the method in terms of realistic examples and specific web platforms to improve the mappings. We will also work on building a user-friendly tool. So far, we have implemented the EMS application using AndroMDA. A part of the future work is to implement the EMS in other platform and expand the existing ones.

References

AndroMDA, www.andromda.org, 15-02-2007

AndroMDA Bpm4Struts Docs, http://galaxy.andromda.org/docs-3.2/andromda-bpm4struts-cartridge/index.html, 20-08-2008

AndroMDA_Forum, http://galaxy.andromda.org/forum/viewtopic.php?t=5884, 12-08-2008

Baresi, L. Colazzo, S. Mainetti, L. and S. Morasca. W2000: A Modeling Notation for Complex Web Applications. In E. Mendes and N. Mosley (eds.) Web Engineering: Theory and Practice of Metrics and Measurement for Web Development. Springer, ISBN: 3-540-28196-7, 2006.

Beydeda, S. Book, M. Gruhn, V. (eds.).Model-driven software development, Berlin ; New York : Springer, 2005.

Botterweck, G. A Model-Driven Approach to the Engineering of Multiple User Interfaces, In: Models in Software Engineering, , Springer Berlin / Heidelberg, Volume 4364/2007, pp. 106-115

Brambilla, M. Comai, S. Fraternali, P. and Matera, M.: Designing Web Applications with WebML and WebRatio. In book: "Web Engineering: Modelling and Implementing Web Applications." Gustavo Rossi, Oscar Pastor, Daniel Schwabe and Luis Olsina. 2007

Cáceres, P. de Castro, V. Vara, J. M. and Marcos, E. "Model Transformation for Hypertext Modeling on Web Information Systems", Proc. ACM Symp. Applied Computing (SAC'06), 2006. pp: 1232-1239.

Ceri, S. Fraternali, P. and Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33 (1-6): 137-157 (2000)

Ceri, S. Fraternali, P. Bongio, A. Brambilla, M. Comai, S. Matera, M.. Designing Data-Intensive Web Applications. Morgan Kaufmann. 2006

Distante, D. Pedone, P., Rossi, G. and Canfora, G. Model-Driven Development of Web Applications with UWA, MVC and Java Server Faces. In: Web Engineering. Volume 4607/2007. Springer Berlin / Heidelberg. Pages 457-472

He, C. Tu, W. and He, K. Role Based Platform Independent Web Application Modeling Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05). Pp: 411-415.

Hibernate, www.hibernate.org, March 27, 2008

Hruby, P. Model-Driven Design using Business Patterns; with contributions by Jesper Kiehn and Christian Vibe Scheller. Berlin ; New York : Springer-Verlag, c2006.
José, M. Cuaresma, E. and Koch, N.: Metamodeling the Requirements of Web Systems. WEBIST (1) 2006: 310-317

Kraus, A. Knapp, A. and Koch, N. Model-Driven Generation of Web Applications in UWE. In Proc. MDWE 2007 - 3rd International Workshop on Model-Driven Web Engineering, CEUR-WS, Vol 261, July 2007

Koch, N. Zhang, G. and Escalona, M. J.: Model Transformations from Requirements to Web System Design Proceedings of the 6th international conference on Web engineering Pages: 281 – 288, 2006.

Koch, N, and Kraus, A.: The expressive Power of UML-based Web Engineering. Proc. Of IWWOST´02, CYTED, pp. 105–119, 2002.

Koch, N. Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process PhD. Thesis, FAST Reihe Software technik, UNI-DRUCK Verlag, December 2000.

Koch, N. Kraus, A. and Hennicker, R. The Authoring Process of the UML-based Web Engineering Approach. In Proceedings of the 1st International Workshop on Web-Oriented Software Technology, 05 2001.

Kroiß, C. and Koch, N. UWE Metamodel and Profile User Guide and Reference, Technical Report, February 2008. Available from (www.pst.ifi.lmu.de/projekte/uwe)

Li, J. Chen, J. Chen, P.: Modeling Web Application Architecture with UML , Proceedings of the 36th International Conference on Technology of Object-Oriented Languages and Systems, 2000. TOOLS - Asia 2000. 265-274

MDA, http://www.omg.org/mda/, October, 1, 2005

Meliá, S. Gómez, J.: Applying Transformations to Model Driven Development of Web Applications. ER (Workshops) 2005: 63-73

Meliá, S. Gómez, J. and Koch, N. Improving Web Design Methods with Architecture Modeling. In 6th International Conference on Electronic Commerce and Web Technologies (EC-Web 2005), Copenhagen,

Denmark, Kurt Bauknecht, Birgit Pröll, Hannes Werthner (Eds.). LNCS 3590, ©Springer Verlag, 53-64, August 2005.

Meliá, S., Gomez, J.: The WebSA Approach: Applying Model Driven Engineering to Web Applications. Journal of Web Engineering, © 5(2), Rinton Press, 121–149 (2006)

Meliá, S. Gómez, J. and Serrano, J. L.: WebTE: MDA Transformation Engine for Web Applications. In: Web Engineering, Volume 4607/2007, Springer Berlin / Heidelberg. Pages 491-495

Muller, P. A. Studer, P. Fondement, F. and Bézivin, J.: Platform Independent Web Application Modeling and Development with Netsilon. Software and System Modeling 4(4): 424-442 (2005)

Netsilon Overview, http://fondement.free.fr/objx/netsilon/, August 5th, 2008

Nikolaidou, M. and Anagnostopoulos, D. A Systematic Approach for Configuring Web-Based Information Systems. In the Journal of Distributed and Parallel Databases, Springer Netherlands, Volume 17, Number 3 / May, 2005 .Pages 267-290

OMG, MDA Guide Version 1.0.1, 12th June 2003

OMG, MOF QVT Final Adopted Specification, November 2005

OMG, Unified Modeling Language: Superstructure, February 2007

Pierantonio, A. Vallecillo, A. Selic, B. and Gray, J.: Special Issue on Model Transformation. Sci. Comput. Program. 68(3): 111-113 (2007)

Rossi, G. Schwabe, D.: Model-Based Web Application Development. In E. Mendes and N. Mosley (eds.) Web Engineering: Theory and Practice of Metrics and Measurement for Web Development. Springer, ISBN: 3-540-28196-7, 2006.

Schmid, H. A. and Donnerhak, O.: The PIM to Servlet-Based PSM Transformation with OOHDMDA, Workshop on Model-driven Web Engineering (MDWE 2005) July 26, 2005

Stahl, T. and Volter, M, Bettin, J. Haase, A. and Helsen, S.: Modeldriven Software Development : Technology, Engineering, Management /translated by Bettina von Stock eth. John Wiley, Chichester, England ; Hoboken, NJ (2006) OMG, MDA Guide Version 1.0.1, 12th June 2003

Tai, H. Mitsui, K. Nerome, T. Abe, M. Ono, K. and Hori, M.: Model-driven development of large-scale web applications, IBM Journal of Research and Development, Volume 48 , Issue 5/6 (September/November 2004) Pages: 797 - 809

UWA Consortium, Ubiquitous Web Applications. In: Proceedings of the eBusiness and eWork Conference 2002, (e2002: October 16-18 2002, Prague, Czech Republic) (2002)

WebML, www.webml.org, May 5, 2008

WebRatio, www.webratio.com, May 6, 2008

Wu, J. H. Shin, S. S. Chien, J. L. Chao, W. S. and Hsieh, M. C.: An Extended MDA Method for User Interface Modeling and Transformation. In: The 15th European Conference on Information Systems. pp 1632-1641 (2007)