ActionUWE: Transformation of UWE to ActionGUI Models

Marianne Busch[1] and Miguel Ángel García de Dios[2]

[1] Ludwig-Maximilians-Universität München
busch@pst.ifi.lmu.de

[2] IMDEA Software Institute
miguelangel.garcia@imdea.org

September 2012

Technical Report 1203
Version 1.0

Research Unit of Programming and Software Engineering (PST)
Institute for Informatics
Ludwig-Maximilians-Universität München, Germany

This work has been supported by the EU-NoE project NESSoS, 256980.

Abstract

Both, UWE (UML-based Web Engineering) and ActionGUI are web engineering approaches for modeling secure web applications. They provide a graphical notation for the representation of the models: a UML profile for UWE and a proprietary notation for ActionGUI. UWE focuses on a high level of abstraction, whereas ActionGUI models can directly be transformed to code. In this report, we describe *ActionUWE*: the model-to-model transformation from UWE into ActionGUI models. As a result, our new ActionUWE approach allows modelers and developers to generate semi-automatically secure code from high-level UWE models.

Contents

1 Introduction

To secure web applications is increasingly important because of rising cybercrime as well as the growing awareness of data privacy. Besides authentication and confidential connections, both data access control and navigational access control are the most relevant security features in this field. However, adding such security features to already implemented web applications is an error-prone task. Therefore, the goal is to include security features in early stages of the development process, i.e., at requirements specification and modeling level.

Existing approaches, such as OOHRIA [6], OOWS [8], WebML [7], UWE [4, 3], or ActionGUI [1] already provide well-known methods and tools for the design and development of web applications. Most of them follow the principle of "separation of concerns" using separate models for different views on the application, such as e.g. content, navigation, presentation and business processes. Most of the available methods do not support the modeling of security, whereas the UWE approach by Koch et al. [3] and the ActionGUI approach by Basin et al. [1] define models for security features like access control. ActionGUI's proprietary notation comprises data models, security models and GUI models, and the whole application logic is represented using OCL. UWE provides a set of UML stereotypes for each view defining a so-called UML profile. UWE's main focus is on the process of discussing and planing an application from different points of view as e.g. requirements, content (data model), navigation, users and roles, basic rights, presentation and process.

At the moment, no model-driven solution for secure web applications exists which can unite the advantages of both approaches, i.e.:

- the advantages of the high-level of abstraction of UWE with its many views (separation of concerns), helping the modelers and developers to plan and implement the application without referring to concrete technologies
- the advantages of a concrete modeling language like ActionGUI which is based on a formal specification of the whole application logic and its access control policies. Furthermore, those policies allow the generation of secure web applications where the security policies are automatically embedded in the GUI.

Our model-driven approach, called *ActionUWE* (using UWE and ActionGUI together) combines both approaches, enabling web engineers to model security issues for web applications in the abstract way of UWE and to transform this representation to ActionGUI. The ActionGUI model has to be enriched with a rather small amount of additional information to specify the details of the application's behavior. Afterwards validation checks that are available for ActionGUI models can be used to examine the model before it is subjected to the model-to-code transformation. Consequently, ActionUWE is the first approach to generate semi-automatically secure software from UWE models.

In this report, we describe ActionUWE, starting with an overview of UWE, ActionGUI and our ActionUWE transformation in section 2. Then we introduce ActionUWE by transforming parts of an example in section 3. Afterwards, the transformation is described in section 4. In order to make this transformation between UWE and ActionGUI possible, the original ActionGUI metamodel is extended (section 5) and some elements are added to the metamodel of UWE (section 6). Finally, we give a summary and an outlook in section 7.

2 The ActionUWE Approach

In this chapter we briefly describe UWE and ActionGUI, before giving an overview of the ActionUWE transformation which converts a UWE model to an ActionGUI Model.

2.1 UWE and ActionGUI Models

As already mentioned in the introduction, ActionUWE aims at comprising the advantages of both, UWE and ActionGUI. UWE is based on the UML standard, i.e., UWE models can be edited with all UML editors that support UML profiles. UWE provides many different models that describe the web application from several abstract points of view. The focus is not on modeling every detail of the application, but on providing an overview of several aspects.

By contrast, the approach of ActionGUI is to model a web application in detail (with a proprietary emf/gmf eclipse plugin) so that a concrete web application can be generated afterwards. For this aim, ActionGUI uses a SecureUML+ComponentUML Model to specify access control rules and defines the whole web application by modeling the GUI enriched with OCL statements defining the application logic.

For the ActionUWE transformation, the following UWE models are useful (cf. left part of figure 1):

The Requirements Model defines requirements for a project and can be used to get a better understanding. However, it is not necessary for ActionUWE itself.

The UWE Content Model contains the data structure, i.e. the data that is used by the application.

The UWE Role Model defines a hierarchy of user groups to be used for authorization and access control issues. It is usually included in a *User Model*, which specifies basic structures, e.g., that a user can take on certain roles simultaneously.

The UWE Basic Rights Model describes the security policy using role based access control. It constrains elements from the *UWE Content Model* and from the *User Model*.

The UWE Presentation Model models graphical parts of the web application.

The UWE Navigational States Model depicts the navigation flow of the application and navigation-related access control policies. Additionally, it comprises a *Navigational Menu Model*, including available menu entries of the application, regardless of their layout.

Further information about UWE models can be found at the UWE website[1].

ActionGUI comprises the following models (cf. right part of figure 1):

The ActionGUI Model contains not only the graphical layout of the application, but also the application logic, which is specified using OCL.

The ComponentUML Model describes the data structure.

The SecureUML Model defines a role based access control policy.

For more detailed information about ActionGUI, see [1].

[1] UWE website. `http://uwe.pst.ifi.lmu.de`

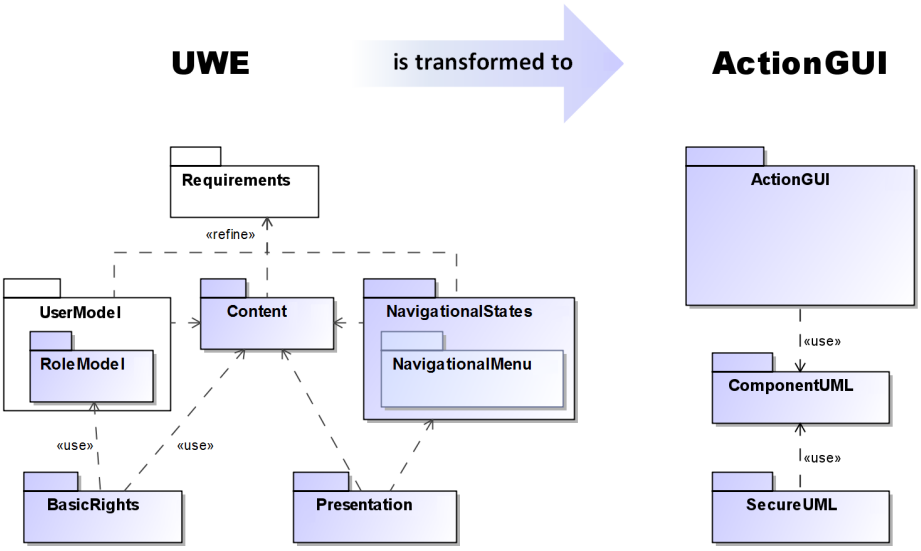


Figure 1: Overview of ActionUWE Transformation

2.2 ActionUWE

The ActionUWE model-to-model transformation relates the UWE model elements to those of ActionGUI in four steps:

Step 1 maps the data structure and creates a main window for the web application and transforms menus modeled with UWE.

Step 2 converts the remainder of the UWE Presentation Model into ActionGUI.

Step 3 transforms the navigational flow information from the UWE Navigational States Model to ActionGUI Model.

Step 4 maps security features.

As ActionGUI and UWE use a different way of grouping features to models, the ActionGUI model itself contains most of the transformed elements:

The UWE Content Model is mapped in a straightforward way to a ComponentUML Model in ActionGUI.

The UWE Basic Rights Model is mapped to a SecureUML Model in ActionGUI.

The UWE Presentation Model is mapped to a set of `Widgets` that are part of an ActionGUI Model.

The UWE Navigational States Model is mapped to certain `Action` and `Event` elements of the ActionGUI Model.

The transformation itself is described in section 4 in further detail. In the following chapter we present our case study *HospInfo* and and use it for illustrating each step of the transformation.

In this document we refer to the metamodel of UWE using the syntax of the concrete UML stereotypes, e.g. we write ≪presentationGroup≫ when referencing the `PresentationGroup` meta class. For the content it makes no difference if you say "an element which is stereotyped by ≪xy≫" vs "an element of the type `xy`".

3 **HospInfo** – introducing ActionUWE by example

In this section, we give an example of how to transform, step by step, an application which is already modeled with UWE into an application modeled in ActionGUI. For this purpose, we work with the `HospInfo` example[2] which is introduced in the following. Afterwards, in section 4, the transformation is specified in general.

3.1 Case Study HospInfo

Our case study, called *HospInf* (Hospital Information), is a prototype of a web-based Hospital Information System. The roles identified for this web application are: visitor, registeredUser, nurse, receptionist, physician and admin. Its main functions are:

- Staff members should be able to register
- The roles of the staff members are set later by an administrator
- Physicians need the permission to create new patient records or change information of patients.
- Nursing staff should be able to read the information about patients from their ward and access the health records of the patients from other wards in cases of emergency (which differs in the fact that the access is logged).
- It should not be the task of physicians to enter organizational patient data, usually this is done by receptionists who can read all information (or at least accounting relevant parts), but cannot change health related data, because they usually have no medical education.

A more detailed description of the example can be found in [2].

In the next sections, *HospInfo*'s UWE models are introduced and transformed to ActionGUI.

3.2 Transformation of the Content Features

In *HospInfo* the focus of interest is on the `Patient`s with some attributes as `name`, `address`, `ward` or `gender` (see figure 2). If the item `administrative` is chosen for the patient's ward from the `Ward` enumeration, it means that a patient has been created for the purpose of software testing.

The UWE Content Model can be transformed to the ComponentUML Model in a straightforward way, because the ComponentUML Model of ActionGUI is mainly just another syntax for classes (UML) / components (ComponentUML Model). The resulting syntax of ComponentUML Model is depicted in figure 3.

3.3 Transformation of Menus

Figure 4 shows the Navigational Menu Model of *HospInfo*. Abstract menu classes as `DefaultMenu` cannot stand for themselves. That means there is no menu that only presents those five menu entries. *HospInfo* offers eight menu entries to an external visitor: the five ones from the `DefaultMenu` and three additional ones that belong to the `Visitor` class.

Regarding the menu structure, the transformation algorithm has to consider e.g. the following elements:

[2]HospInfo UWE example. http://uwe.pst.ifi.lmu.de/exampleHospInfo.html

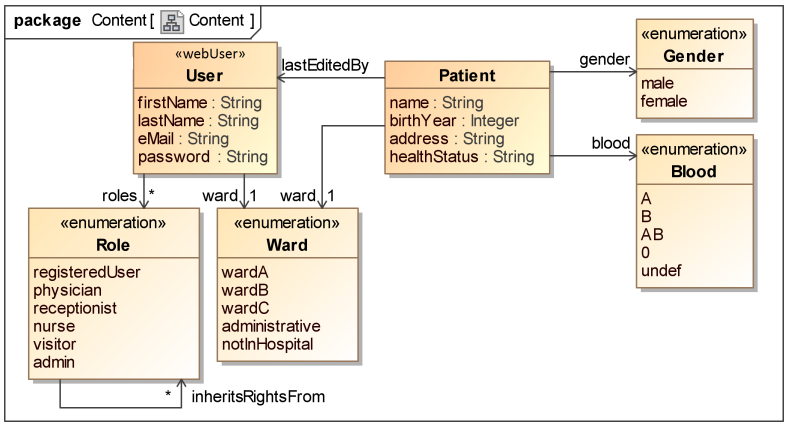


Figure 2: UWE: Content Model

- ≪navigationMenu≫ classes that are connected to the Presentation Model by tags as labeled in figure 4.
- The property `ShowPatients` can exemplarily be found in the UWE Presentation Model as shown in figure 5(a).
- It is contained by the ≪presentationMenu≫ class with the name `NavigationMenu`. Figure 5(b) depicts the children of this class in the containment tree of the CASE tool MagicDraw[3].
- Each of those classes (that are also added as UML property to the `NavigationMenu` class) are transformed to menu entries in ActionGUI.

An excerpt of the resulting ActionGUI model is depicted in figure 6. Please notice that only some classes are transformed as proof of concept.

3.4 Transformation of the UWE Presentation Features

Figure 7 depicts the default page layout of *HospInfo*: a `Headline` ≪text≫[4] on top, a `Footer` at the bottom of the page and on the left a `NavigationMenu` with anchors that determine the body of the `ContentArea`, ≪presentationAlternative≫ class, which means that exactly one of the included ≪presentationGroup≫ properties can be displayed simultaneously. Furthermore, some details of the log-in form are shown (lower right), which are stereotyped by ≪inputForm≫ to denote that the user can submit credentials to a Web server

In ActionGUI the UWE classes and properties are recursively replaced by corresponding ActionGUI elements as shown in figure 8. The algorithm starts at the `DefaultPageLayout` class and then transforms e.g. the `ContentArea` and the nested property for the `LogInPage` form (figure 9).

3.5 Transformation of the UWE Navigational States

Figure 10 shows possible ways to navigate in *HospInfo* inside the area for receptionists or physicians. Navigation means to browse through the

[3]MagicDraw. https://www.magicdraw.com
[4]UWE stereotype names & symbols: `http://uwe.pst.ifi.lmu.de/profileOverview.html`

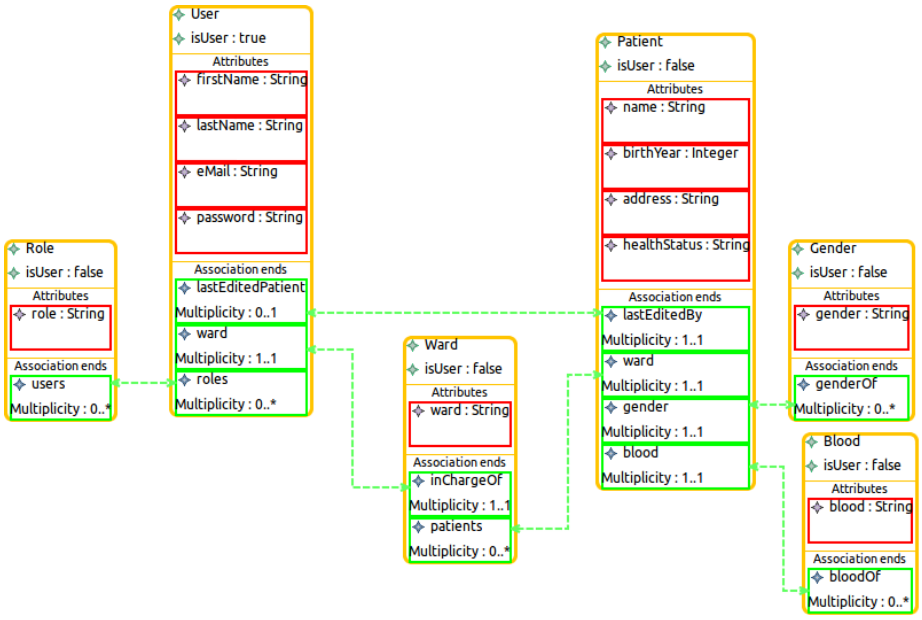


Figure 3: ActionGUI: ComponentUML Model

application, thus the diagram specifies how parts of a web page change. The transitions on the left stand for menu entries, as clicking on a menu entry will fire the corresponding transition. Transitions starting on the border of a state are specified in UML to leave the state and enter again.

This paragraph explains to the interested reader, how those menus are linked with the main state machine, which is depicted in figure 16. Because of the integrated menus, shown in 16 the three triggers `createPatient()`, `showPatients()` and `searchPatients(...)` (see figure 10) are removed from the inner transitions and combined with the first transition in the `MultipleRolesArea` state, as defined in section 4.4.2. After applying the transformation according to the referenced definition, three transitions connect the `MultipleRolesArea` state with three new entry points of the topmost substate machine.

In the following, we show an example of how to map navigation flow of the UWE Navigational States Model to the ActionGUI model. The basic idea is to iterate over UWE Presentation elements, to take into account the corresponding states and transitions in the UWE Navigational States Model and to enrich the ActionGUI Model with (conditional) `OnClick` actions.

In the UWE Navigational States Model, the navigation flow is modeled by transitions between different states, but transitions do not always directly connect a source state to a target state. `Choice Pseudostate` elements can appear in a transition to split the transition into different branches depending on the conditions of these `Choice Pseudostate` elements.

The transition `edit` (surrounded by the lower rounded shape in figure 10) is an example of a transition without `Choice Pseudostate` elements. This transition is mapped to ActionGUI Model as shown in figure 11. On the other hand, the transition `finish` (surrounded by the up-

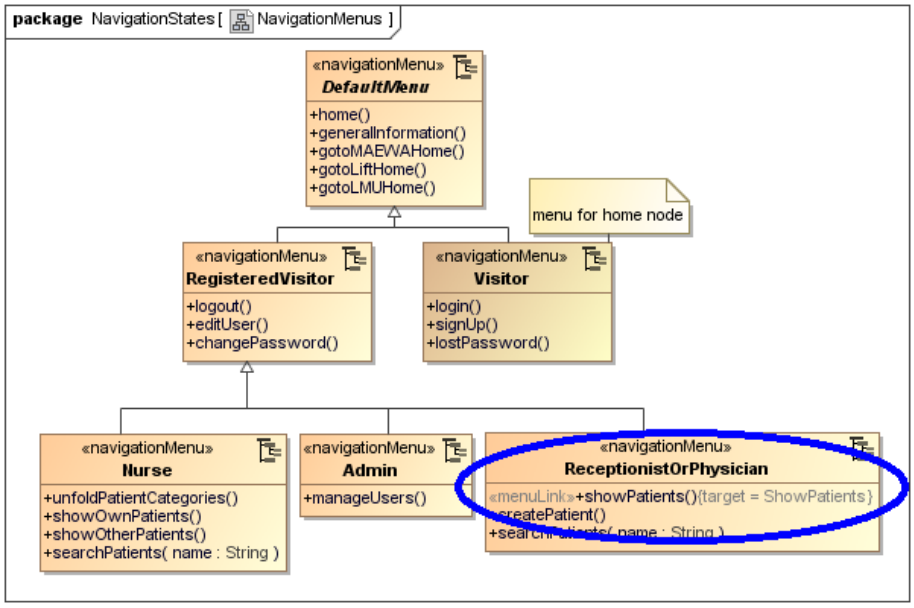
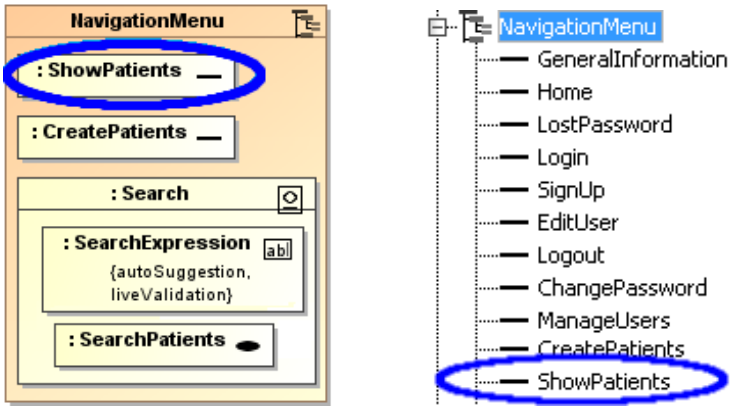


Figure 4: UWE Navigation menu



(a) HospInfo: menu (diagram) (b) HospInfo: menu (tree)

Figure 5: Excerpt of the UWE Presentation Model

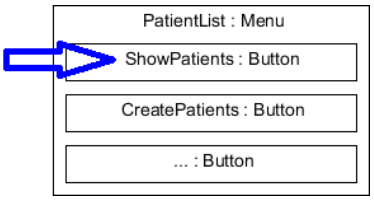


Figure 6: ActionGUI: Part of the navigation menu

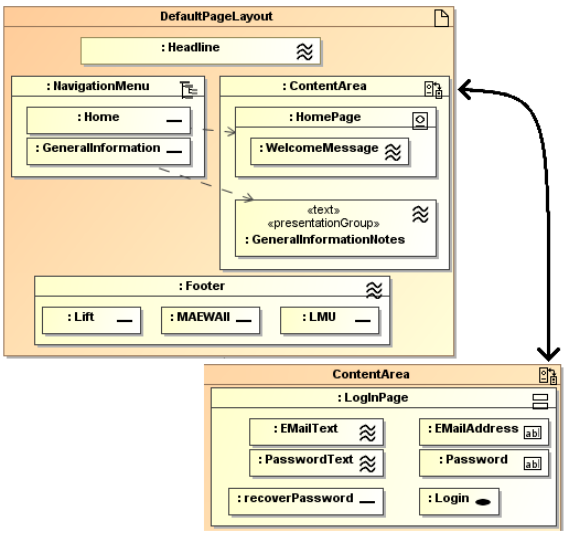


Figure 7: Example of UWE presentation diagrams

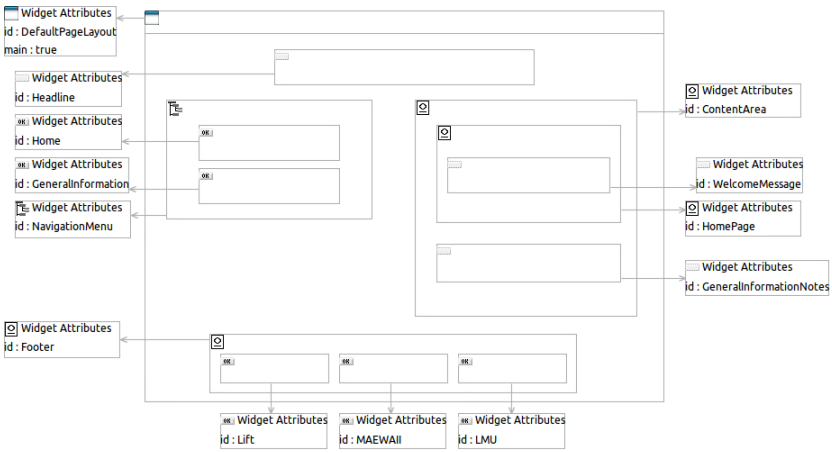


Figure 8: Default layout transformed in ActionGUI

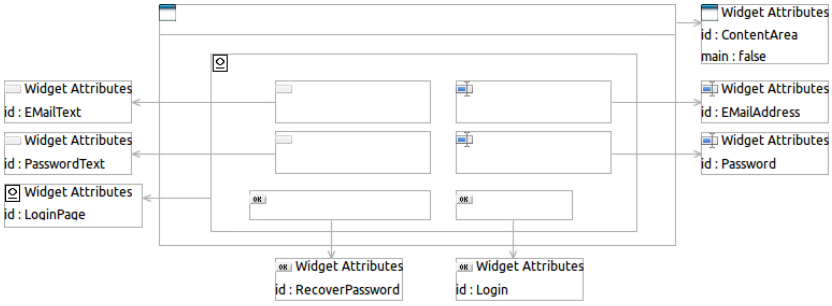


Figure 9: Log-in form transformed in ActionGUI

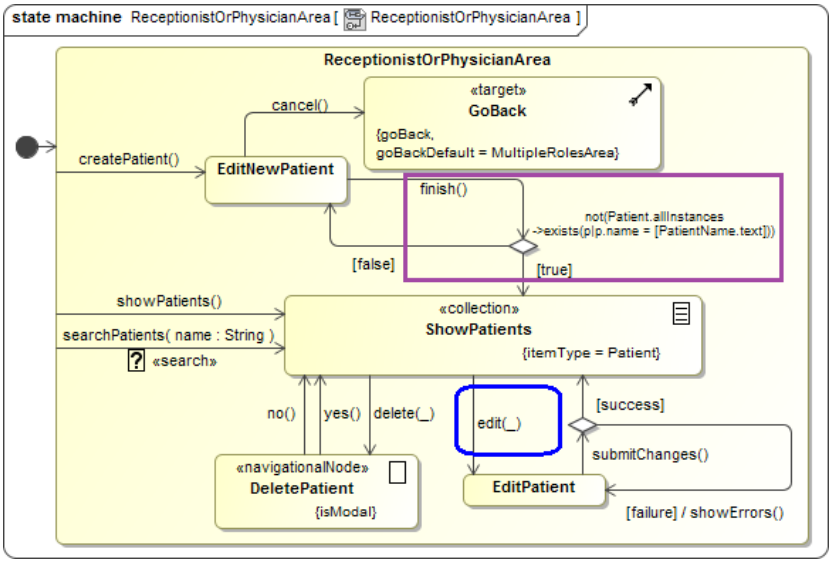


Figure 10: Two transitions of the UWE Navigational States Model

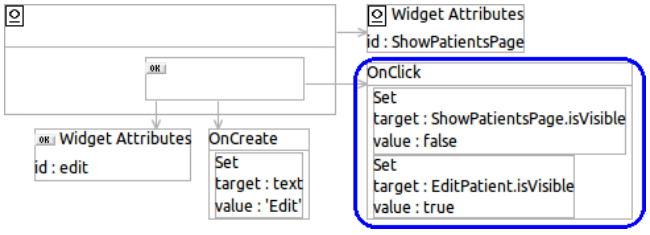


Figure 11: ActionGUI: Simple transition mapped to ActionGUI

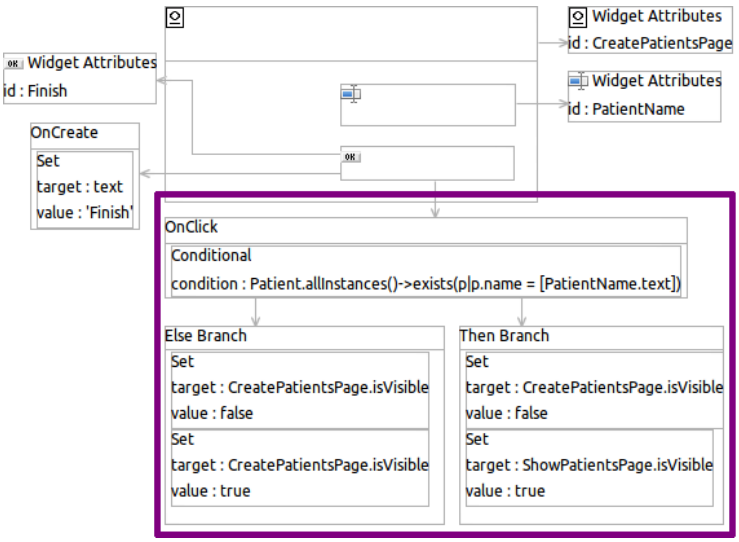


Figure 12: ActionGUI: Complex transition mapped to ActionGUI

per box) serves as an example of a transition with a `Choice Pseudostate` element. Figure 12 depicts how this transition is mapped to ActionGUI. In this case, a `Conditional Action` is added to the event in order to implement the condition of the `Choice Pseudostate` element.

For this transformation it is necessary to know which UWE ≪presentationGroup≫ is shown in a certain state (≪form≫ inherits from ≪presentationGroup≫) and which transition is triggered from an ≪interactiveElement≫ like a ≪button≫. Figure 13 shows a simple example which is linked to states / transitions which have been depicted in figure 10.

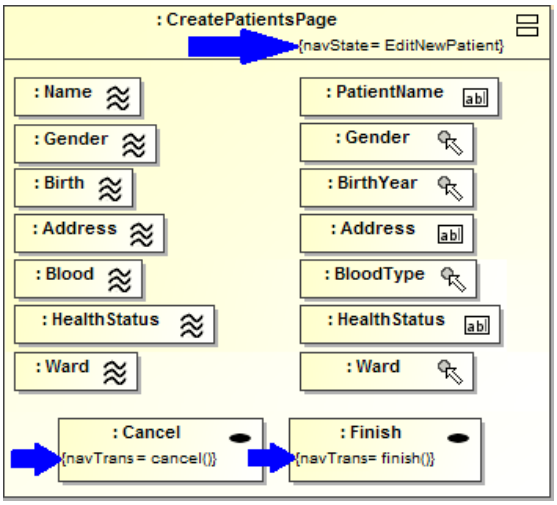


Figure 13: UWE's Presentation Model linked to Navigation States Model

Furthermore, the transformation algorithm has to remember the connections from the UWE Presentation Model to the ActionGUI Model which have been established in the previous steps.

3.6 Transformation of UWE's Security Features

In this section, we give several examples of how to map the different security features from the UWE profile to ActionGUI.

3.6.1 Basic Rights Model

An application modeled with UWE can include security policies. The UWE Basic Rights Model is used to define which roles exist and which permissions over the application data those roles have.

Figure 14 shows the UWE Basic Rights Model that corresponds to the security policy of the HospInfo example. There are six roles, four of them (*admin*, *receptionist*, *physician* and *nurse*) are sub-roles of the super-role *registeredUser*. Each role has its own permissions on the application data, which has been defined in the UWE Content Model. Some permissions have additional *authorization constraints* associated to them.

In step 4 of section 4, we explain the mapping of the UWE Basic Rights Model to the SecureUML+ComponentUML Model in further detail; right now, we give some examples.

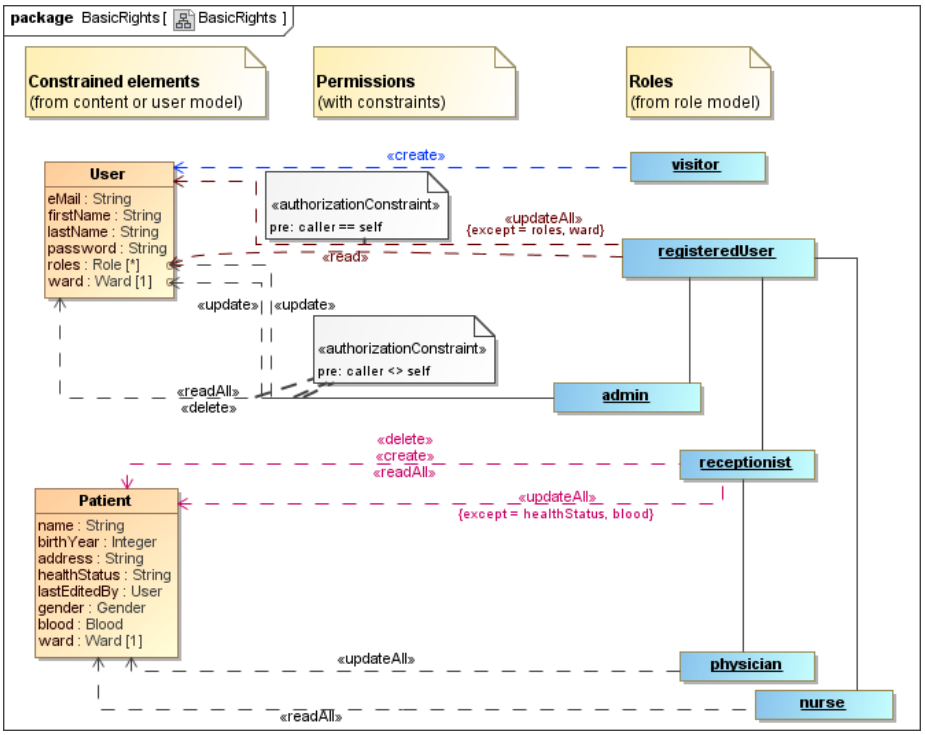


Figure 14: UWE BasicRights Model

Figure 15 shows the resulting SecureUML Model. Each role of the UWE Basic Rights Model is directly mapped to a role in the SecureUML Model, and also the hierarchy between them.

In figure 14, the ≪updateAll≫ permission between the role *physician* and the class *Patient* is mapped in the SecureUML Model to a permission named *updatePatientsPhysician*, with an *Patient EntityUpdate* action attached. This is an example of how composite actions are transformed: If there is a permission with a composite action and an ≪except≫ restriction in the UWE Basic Rights Model, the corresponding permission in SecureUML Model will contain atomic actions for every attribute and association of the corresponding class, except for those listed in the ≪except≫ restriction. This is the case for the *updatePatientsReceptionist* permission: it contains atomic update actions for each allowed element, instead of containing just one composite action for every element.

The ≪delete≫ permission of the role *admin*, in figure 14, additionally has an attached ≪authorizationContraint≫ which defines that the user which should be deleted has to be different from the administrator that executes the action. This permission is mapped, with the corresponding authorization constraint, to the permission *deleteUser* in the SecureUML Model. Remaining formal permissions are transformed in the same way.

The slightly extended OCL language is used to define the authorization constraints in the UWE Basic Rights Model as well as in ActionGUI's SecureUML Model defines the keywords *caller* and *self*. The first refers to the current user of the application whereas the latter references – as is usual – the object on which the permission is applied.

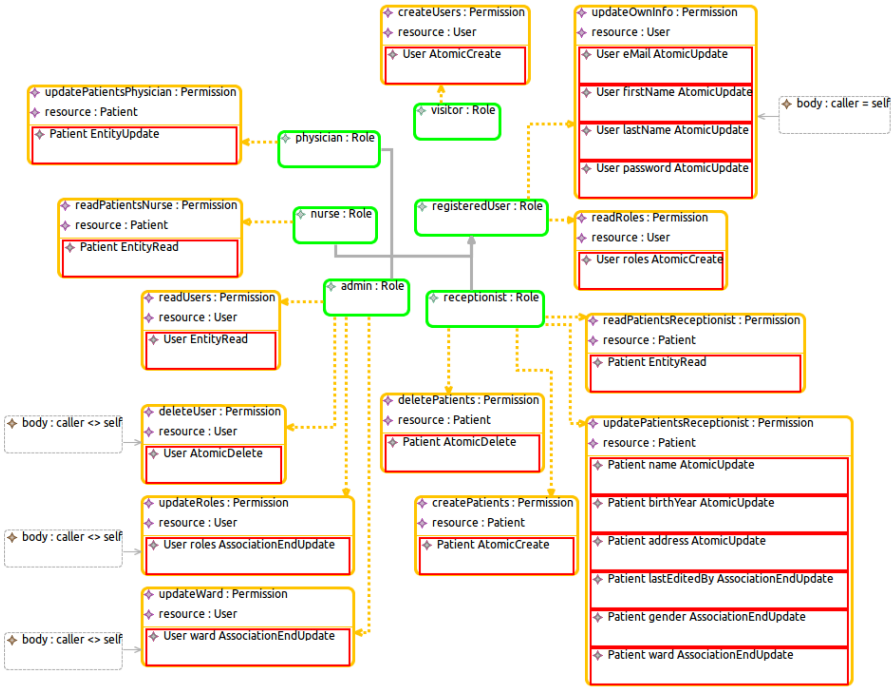


Figure 15: ActionGUI's SecureUML Model

3.6.2 Security Aspects of Navigational States

In the UWE Navigational States Model, several security aspects are taken into account: secure communication links, authentication and navigational access control. Secure communication links cannot be mapped to ActionGUI, authentication patterns are available in the UWE profile and if they are used, they can be translated without further ado. Navigational access control has already been partly transformed when we had a look at the Choice Pseudostates in section 3.5. Additionally, we have to clarify how guards on menu entries can be transformed to ActionGUI so that the user cannot see a prohibited menu entry. Therefore, we give an example of how a menu transition is mapped to the ActionGUI Model in this section.

figure 16 illustrates the ≪updateMenus≫ transition *tr* surrounded by the rounded box on the left. It is fired when a user logged-in successfully. In the other circle, a transition is shown which is a typical menu transition which leaves the state first and enters again, targeting a substate.

The stereotype ≪integratedMenu≫ is just a simplifying abbreviation so that we do not have to model all menus for the admin (that are hidden in the substate machine) independently. Further details can be found in section 3.5. In the *HospInfo* example it is important that the menu entries for the administrators are only available when the set of roles for current user includes the admin role. In case the user is a physician and an administrator at the same time, all those menus are shown when he or she is logged-in.

For our transformation to ActionGUI, we pick the ≪updateMenus≫ transition *tr* and have a look at the "Login" button which is used after having entered the credentials. After our previous transformations, the "Login"

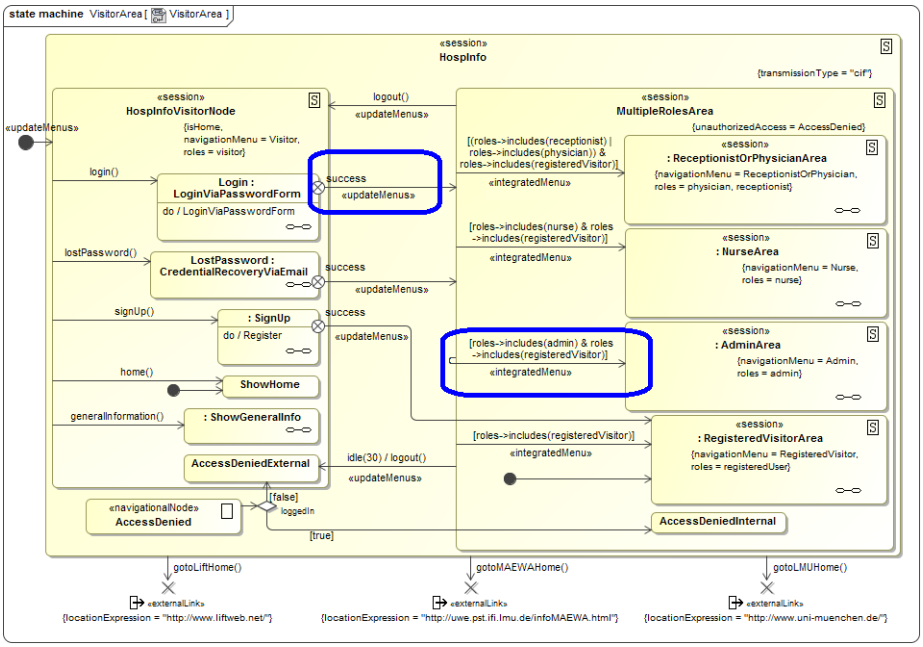


Figure 16: Example of ≪updateMenus≫ transitions in the UWE Navigational States Model

widget should have a "OnClick" ActionContainer attached that already hides a kind of `VisitorArea` panel and shows an `MultipleRolesInternal` panel.

Furthermore, we iterate over at all menu transitions (selected according to their connection from the UWE Presentation Model) that cannot fire in the target state of the ≪updateMenus≫ transition, which is the `MultipleRolesArea`. Those are e.g. `login()` or `lostPassword()`, thus we hide them in the ActionGUI menu.

Menu transition that are reachable from within the `MultipleRolesArea` are for instance the nurse menus, but they are guarded and only available for nurses, not for our example-user which is physician and admin. Consequently, the guard evaluates to false and the nurse menus are hidden for our user.

Besides, there are the menus for registered visitors, for physicians and for administrators which should be displayed. The hiding or showing of menu items in ActionGUI are done using a `Set` which sets the variable `isVisible` of the menu widgets, as shown in figure 17. In our example, `roles` serves as a short form for `caller.roles`. Finally, the non-guarded `logout()` menu transition is reachable as well and should also be available in the menu.

Note that according to the definition of the ≪integratedMenu≫ stereotype, all menus that are modeled within substate machines have to be taken into account, as e.g. `CreatePatient` which has been depicted in figure 10.

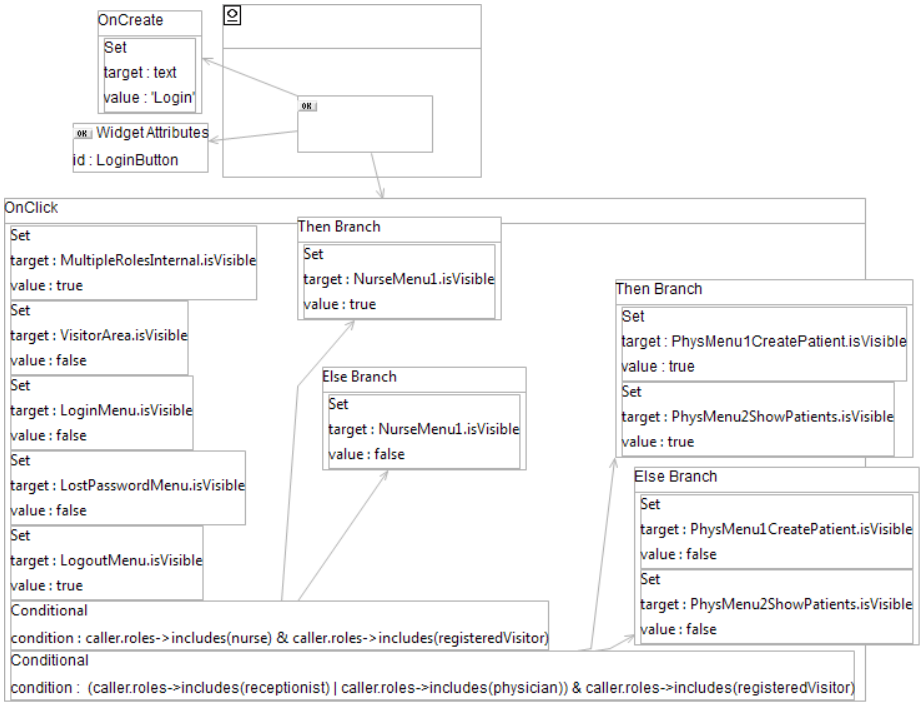


Figure 17: ActionGUI: Part of the transformed «updateMenus» transition

4 Transformation Process

As already described in section 2.2, the transformation process from UWE to ActionGUI is performed in in four steps: step 1 creates a main window for the web application and transforms available menus that are modeled with UWE. Step 2 transforms the remainder of the UWE Presentation Model into ActionGUI. Afterwards, step 3 transforms the navigational flow information from the UWE Navigational States Model to ActionGUI Model. Finally, in step 4 security features are mapped.

Our goal is a web application app', which is partially modeled in ActionGUI. The whole data structure of *app* should be mapped. The layout and the navigation flow (including security features) of the application is also mapped, but the behavior of the application cannot be transformed since it is only partly defined through Activity diagrams and in this approach, we do not take Activity diagrams into account.

Generally, for each element that is transformed to a `Widget`, an unique `id` has to be generated and stored as an attribute, or copied in case an ≪uiElement≫ {id} is specified in UWE.

At the beginning we directly translate the UWE Content Model in the representation of ComponentUML Model, which is easy, as both focus on elements, attributes, associations and operations (although operations are not needed for our transformation right now). An example can be found on the previous pages (section 3).

4.1 Step 1: Mapping of Navigational Menus

First an outermost `Window` has to be created in ActionGUI with the attribute `main` set to `true`. Within the Navigational Menu Model, for each operation of all classes, the corresponding ≪anchor≫ class from the UWE Presentation Model has to be added to a new set M. This corresponding classes are linked using the ≪menuLink≫ {target} tag. If it is not set, the menu entry is ignored, i.e. not added to M.

For each element in M, the outermost parent class (that contains the property of the element) stereotyped by ≪presentationMenu≫ has to be found and added to a new set called P. All this is necessary because the menu is not modeled at one place in the UWE Presentation Model.

Afterwards, the following `menuTransformation`-function is used, with actual parameters (menu m, set P), where m is a newly created menu that is added to the outermost ActionGUI window of the application, which has been created above.

```
menuTransformation(Class z, Set<Class> P){
 for (classes c : P){
  if (Class c is stereotyped by presentationMenu){
   create an ActionGUI Menu with the name of c
   add that menu to z
   call menuTransformation(c, direct descendants of c)
  }else if (Class c is stereotyped by anchor){
   create a button with the name of c
   add it to the menu z
  }
}
```

Additionally, we remove anchors or menus from the UWE Presentation Model that are not related to an operation in a ≪navigationMenu≫.

4.2 Step 2: Mapping of Presentation Elements

This step consists of creating further graphical elements of the application and defining their nested structure. In the following we describe how UWE modeling elements are mapped to ActionGUI elements.

- Given the UWE Presentation Model, each ≪presentationPage≫ is mapped to a `Window` (i.e. for each ≪presentationPage≫ a `Window` is created in ActionGUI).
- Each ≪presentationGroup≫ excluding the ≪navigationMenu≫ ones, is mapped to an ActionGUI `Panel`. Likewise ≪inputForm≫, ≪presentationAlternatives≫ and ≪tab≫ are mapped to `Panel`s.
- ≪anchor≫ and ≪button≫ are mapped to `Button`.
- ≪text≫ is mapped to `Label`.
- ≪textInput≫ is mapped to `TextField`.
- ≪selection≫ is mapped to `Table`, if {multiple=true}. Otherwise it is mapped to `ComboBox`.
- ≪slider≫ is mapped to `ComboBox`.
- ≪iteratedPresentationGroup≫ is mapped to `Table`. Eventually add a `Panel` for each row to describe more complex information.
- Other elements of the UWE Presentation Model can not be mapped automatically to ActionGUI yet (e.g. ≪image≫, ≪mediaObject≫, ≪imageInput≫, ≪fileUpload≫ and ≪customComponent≫). However, the transformations can be added as soon as ActionGUI supports those kinds of widgets.

The recursive algorithm for mapping the UWE presentation elements to the ActionGUI elements is the following: collect all ≪presentationPage≫ s and within each page look for ≪uiElement≫ s. For each ≪uiElement≫ create the corresponding ActionGUI element specified above and add it to the corresponding ActionGUI `Window` (that stands for the ≪presentationPage≫). This means that we instantiate the association between `Container` and `Widget` and specify the `container` and `content` roles accordingly. For each ≪presentationGroup≫ that is included inside the newly transformed element recursively transform the contained properties to nested ActionGUI elements.

A constraint (precondition) is that in UWE no ≪presentationPage≫ is included in other states as a property.

4.3 Step 3: Mapping Navigation Features

This step consists of mapping the navigation flow of the UWE Navigational States Model to the ActionGUI model. This means navigation links between the different elements of the application are created by using the `ConditionalAction` and `Set` actions of ActionGUI.

We expect every element of the UWE Presentation Model stereotyped with ≪presentationGroup≫ (or subtypes) to be linked to a state of the UWE Navigational States Model by the tag {navState : State}. Furthermore, every element of the UWE Presentation Model stereotyped with ≪interactiveElement≫, must be linked to a transition of the UWE Navigational States Model using the tag {navTrans : Transition}, in order to specify which element of the UWE Presentation Model triggers which transition. (To be exact, the UWE Navigational States Model has to be transformed into a version without UWE stereotypes and tags [2] like ≪navigationalNode≫ {isModal}, but for this first version of ActionUWE, we neglect those subtleties, because at the moment ActionGUI does not support features like modal windows or the navigation to a certain point

within the application using a URL. Nevertheless, ≪integratedMenu≫ s have to be kept in mind, as explained in further detail in section 4.4.2 and ≪target≫ {goBack} may be transformed to "OnClick/Back" in ActionGUI.)

In addition, we expect every `Choice Pseudostate` element *cp* of the UWE Navigational States Model to be defined in the following way:

- Exactly one transition acts as an entry of the *cp*.
- One or two outgoing transitions of the *cp*.
- The condition has to be written using the OCL language and must be specified for *cp*. If there is no guard (for example when the success of an action is checked informally), the [true] or [success] branch is taken and transformed adding just a comment in ActionGUI to remind the developer of dealing with the [false] or [failure] branch.
- If the *cp* has two outgoing transitions, they must be guarded by `[true]` and `[false]` referencing the `then` and `else` branch respectively.
- If the *cp* has just one outgoing transition, this branch references to the `then` branch and it can be optionally guarded by `[true]`.

Given a transition *tr* of the UWE Navigational States Model, we call *ta* the target state of the transition *tr*. If the target state *ta* is not referred by any ≪presentationGroup≫ element in the UWE Presentation Model, then there must be a state contained in *ta* that has a starting point. In this case, this state will be the target state *ta*.

Furthermore, there is an ≪interactiveElement≫ element of the UWE Presentation Model that triggers the transition *tr*. To make things easier the current version of ActionGUI only considers one overall menu and one panel visible at the same time (boolean *isVisible*). As usual, each `Panel` is a `Container` for several elements.

For each transition *tr* of the UWE Navigational States Model, which is referenced from the UWE Presentation Model, the mapping algorithm to the ActionGUI Model is the following:

- Create an `OnClick` event *ev* and link it to the corresponding ActionGUI widget *w*. (This is the widget which has been created from a UWE ≪interactiveElement≫ which is linked to the transition *tr*. It can also be a menu entry.)
- Call the function `createNavFlow(w, tr, ev)` defined below:

```
createNavFlow(Widget w, Transition tr, ActionContainer ac){
 if (the target of tr is a state){
  create a Set action s1 in ActionGUI
   and set isVisible of the source_panel(w) to false;
  link s1 to the ActionContainer ac;
  create a Set action s2 in ActionGUI
   and set isVisible of the target_panel(w) to true;
  link s2 to the ActionContainer ac;
 }
 else if (the target of tr is a Choice ch){
  create a ConditionalAction ca in ActionGUI;
  get the condition c of the Choice's target;
  set the condition attribute of ca to c;
  link the ConditionalAction ca to ac;
  for (outgoing transition et of ch){
   if (et is not guarded || et is guarded by [true]){
    create a ThenContainer tc in ActionGUI;
```

```
    link the tc to ca;
    call createNavFlow(w, et, tc);
   } else if (et is guarded by [false]){
    create an ElseContainer ec in ActionGUI;
    link ec to ca;
    call createNavFlow(w, et, ec);
   }
  }
 }
}
```

`source_panel(w)` refers to the ActionGUI panel which has been transformed from a UWE Presentation element containing the ≪interactiveElement≫ (the counterpart of w) that is connected to the transition tr.

`target_panel(w)` refers to the ActionGUI panel which is going to replace the `source_panel(w)`. It has been transformed from a UWE presentation element which is connected to a state in the UWE Navigational States Model that is the target of the transition tr.

As already stated in the introduction of this section, transitions that are not mapped to the UWE Presentation Model are ignored as their functionality has to be modeled in ActionGUI with greater detail. Usually not many transitions are modeled in the UWE Navigational States Model that do not alter the GUI of the applications (but change the state of the application itself).

4.4 Step 4: Mapping Security Features

In this step, the security-awareness from the UWE Basic Rights Model and the UWE Navigational States Model is added to the ActionGUI model. First we transform the RBAC structure, then the menu is secured and navigational access control features are mapped.

4.4.1 Basic Rights Model

The UWE Basic Rights Model is just another graphical representation of the SecureUML+ComponentUML language that is used in ActionGUI (in a version that differs from the original which is described by Lodderstedt et al. [5]).

- For each `Role` instance r from the UWE Role Model, create the corresponding instance r' of the `Role` class in the SecureUML Model.
- For every inheritance relationship between two roles, $r1$ ans $r2$ in UWE Basic Rights Model, create a link between the corresponding two roles, $r1'$ and $r2'$ in the SecureUML Model.
- For each set of dependencies d between a `Role` instance r and a constrained class c from the UWE Role Model, create a `Permission` instance p in the SecureUML Model. Set the resource of p with the entity e in the ComponentUML Model which corresponds to c in UWE Role Model by creating a link between p and e. Also, create a link between p and r'. If a subset of dependencies is constrained by a set of ≪authorizationConstraint≫ s, then create an own `Permission` instance for them.
- The allowed actions are mapped in the following way:
  1. For all ≪create≫ and ≪delete≫ dependencies between a `Role` instance and a class in the UWE Basic Rights Model, `EntityCreate`

and `EntityDelete` actions are added to the corresponding permission in SecureUML Model.

2. For all ≪read≫ and ≪update≫ dependencies between a role and an attribute, `AtomicRead` and `AtomicUpdate` actions are created respectively, and linked to the corresponding permission.
3. The ≪readAll≫ and ≪updateAll≫ dependencies without {except} tag specified are mapped to `EntityRead` and `EntityUpdate` compound actions respectively. Dependencies with {except}-tag are transformed executing the previous step for every attribute/role of the constrained class, except for the ones described in the {except} tag.
4. ≪execute≫ and ≪executeAll≫ dependencies cannot be translated at the moment, because ActionGUI does not support them in the same manner.

- Map all authorization constraints in a straightforward way and connect them to the newly created `Permission` classes.

4.4.2 Security Aspects of Navigational States

In the UWE Navigational States Model, several security aspects are taken into account:

Secure Communication Links (ensure e.g. confidentiality, integrity and freshness). They cannot be transformed to ActionGUI.

Authentication in terms of predefined patterns available in the UWE profile. They can be transformed by adding them to the UWE Presentation Model before step 3.

Navigational Access Control as described below.

There are several ways to address Navigational Access Control. The most important ones can be transformed as described in the following sections.

Showing and Hiding Menus according to permissions. In UWE, menus are common transitions that usually leave a state, enter again and point to a substate, so that it can be fired from any substate.

To ease the modeling of menus that are modeled in detail in substate machines, the ≪integratedMenu≫ stereotype allows the modeler to split one menu into several submachine states without using entry points for each menu. This comes in handy if many roles and many menus are used.

For a full understanding of this stereotype, the constraint and definition of a transformation to plain UML without stereotypes, are cited from Busch [2]:

> **Constraint.** A composite state s contains m submachine states $p_i \in P$ $(i = 1, \ldots, m)$. P is the set of submachine states, where transitions stereotyped by ≪integratedMenu≫ end. Transitions in the set T are typed by ≪integratedMenu≫ with the guard expressions g_i and no other properties (like effects or triggers) and each $t_i \in T_i$ connects s or an initial node – see also the paragraph below – with p_i. Each p_i refers to a state machine p'_i. Each p'_i contains o_i transitions $t'_{ij} \in T'_i (j = 1, \ldots, o_i)$ from a composite state s'_i (which is the target of an initial state) in p'_i to a substate s''_{ij} in s'_i.

An initial state can exist in s. If no other transition is connected with a submachine state p_i that includes menu transitions, which should be integrated, the transition coming from the initial node, pointing to p_i, has to be stereotyped by ≪integratedMenu≫.

Definition. For each t'_i, the transition t_i that targets p_i is copied, which results in the transformed transitions $\hat{t}_{ij}$ with the guards g_{ij}. Each $\hat{t}_{ij}$ targets a new entry point e_{ij} of p_i with the corresponding representation e'_{ij} in p'_i. All t'_{ij} are transformed to $\hat{t}'_{ij}$ that start at e'_{ij} instead of s'_i.

For all transitions $\hat{t}_{ij}$, all properties of $\hat{t}'_{ij}$ are added. The guard g_i is added with '&' to the guards g_{ij} (for $j = 1, \ldots, o_i$). If one guard is empty, the '&' is left out. If a t_i starts at an initial node, the ≪integratedMenu≫ stereotyped is removed, if not, t_i is deleted itself.

All transitions without the ≪integratedMenu≫ stereotype remain unchanged.

This transformation has to be considered whenever dealing with the UWE Navigational States Model. An example is shown in figure 18 (and explained in further detail in [2, p.51]).

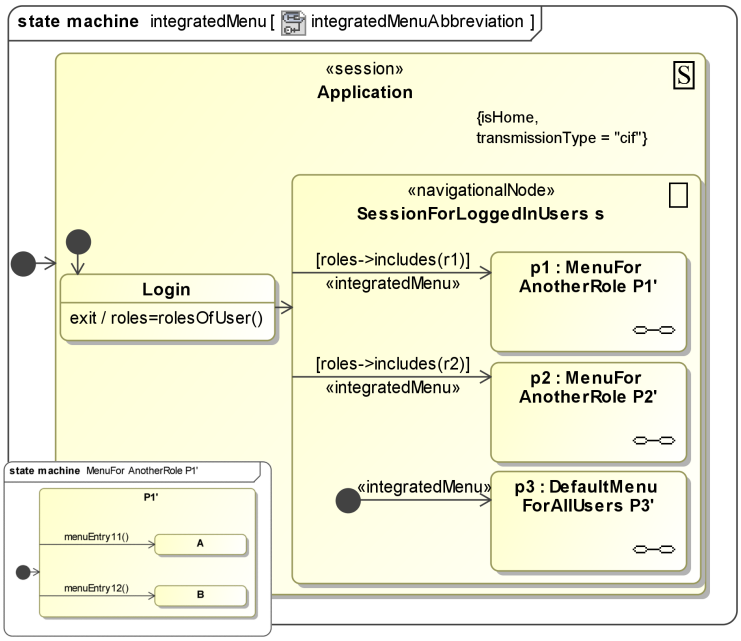


Figure 18: UWE: HospInfo Example of an ≪integratedMenu≫ (from [2, p.52])

For showing or hiding menus in UWE, we have to take into account transitions which are *related to menu elements* from the UWE Presentation Model, since the actions of showing and hiding menus are executed when those transitions are triggered: every time a transition stereotyped with ≪updateMenus≫ is triggered, all menu entries are hidden and just the ones that correspond to the menu transitions reachable from target state, are shown. We added the ≪updateMenus≫ stereotype to be able to avoid recalculating the menu for each transition.

The rough behavior of showing and hiding menus is the following: When a transition *tr* stereotyped with ≪updateMenus≫ is triggered, then: For each menu transition *mt*, we will check if the corresponding menu entry should be shown or not:

- Hide menu entries that cannot be accessed from the target state *s* of *tr*.
- If *mt* is accessible and does not have any condition or a condition that evaluates to `true`, then show the menu that corresponds to *mt*.
- If *mt* is accessible and the condition evaluates to `false`, do not show the menu entry.

As we have already discussed in step 3, all transitions are mapped to an `ActionContainer` element in the ActionGUI Model which contains certain actions, depending on whether the target of the transition is a `Choice Pseudostate` element or a `State` element in the UWE Navigational States Model. We expect all transitions stereotyped by ≪updateMenus≫ to directly lead to a state, thus these transitions are already mapped to ActionGUI `ActionContainer` elements containing the action of hiding the current panel and showing the target panel.

Given a transition *tr* stereotyped with ≪updateMenus≫, and given the corresponding (On-Click) `ActionContainer` *ac* that maps *tr* to the ActionGUI Model, the concrete algorithm of mapping the behavior of *tr* is called as: `updateMenus(tr, ac)`.

```
updateMenus(Transition tr, ActionContainer ac){
s = the state that is reached by tr;
for (menu widget me in the ActionGUI model)
 mt = the menu transition that corresponds to me;
 if (mt is not reachable from s){
   create a Set action se
    and set isVisible of me to false;
   link se to ac;
 } else if (mt is reachable from s and not guarded)
   create a Set action se
    and set isVisible of me to true;
   link se to ac;
 } else if (mt is reachable from s and guarded){
   create a ConditionalAction ca in ActionGUI;
   get the guard g;
   set the condition attribute of ca to g;
   link the ConditionalAction ca to ac;

   create a ThenContainer tc in ActionGUI;
   link the tc to ca;
   create a Set action se in ActionGUI
    and set isVisible of the me to true;
   link se to tc;

   create an ElseContainer ec in ActionGUI;
   link the ec to ca;
   create a Set action se in ActionGUI
    and set isVisible of the me to false;
   link se to tc;
 }
}
```

Note that menus can be nested and if no inner menu is shown the parent should also be hidden.

Accessing States. How the access to states and their substates is additionally guarded in UWE in order to allow to use a set of URLs in the final web applications that target different states without causing security problems by guessing or saving those URLs. As specified in the UWE profile, the {role} tag that specifies roles which are allowed to access a state is internally transformed into a {roleExpression} tag which then is transformed to a choice for the (usually not modeled) transition that comes from a location choice just behind the overall initial pseudostate. What to do in case of unauthorized access is modeled using the tag ≪unauthorizedAccess≫ that references another state.

In ActionGUI only one URL for the hole application is allowed, therefore this kind of access control does not have to be transformed. However, if the access to certain actions are not allowed, an "access denied" pop-up appears in the resulting web application.

Automatic Logout. To automatically logout is modeled in UWE using a `logout()` transition e.g. `idle(mins) / logout()`. In ActionGUI, there are two ways of dealing with the log-out operation automatically. The first one is that, by default, generated applications with ActionGUI execute automatically the logout operation after a certain seconds of inactivity. The second one is that you can define a system operation in the ComponentUML Model of your application that implements the log-out feature. This operation could be called from any part of the ActionGUI Model through the OCL language.

5 Extension of the ActionGUI Metamodel

The current approach extends the ActionGUI metamodel from from Clavel et al.[5] with two meta-classes:

- `Panel` inherits from `Container`
- `Menu` inherits from `Container`, but it can only contain elements of type `Menu` or `Button`. `Menu` elements can only be contained in elements of type `Window`, at most one menu for every window. A variable called `foldAutomatically : Boolean` is added for collapsing open menu trees automatically in case of selecting other menu entries.

The class `Widget` needs an additional variable called `isVisible : Boolean` that specifies whether it is displayed or not.

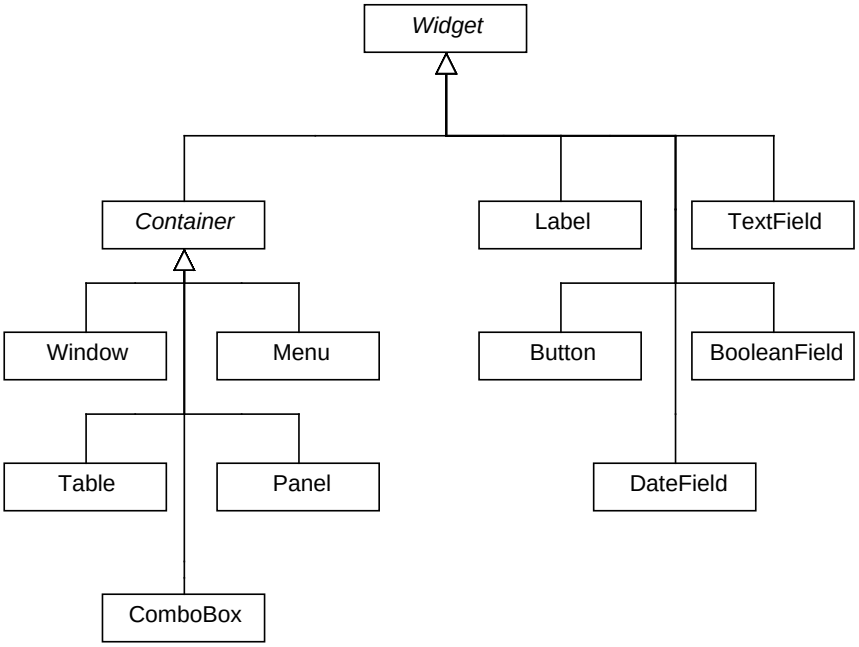


Figure 19: Part of the ActionGUI Metamodel

[5] ActionGUI metamodel v. 1.2012 personal communication

6 Extension of the UWE Metamodel

In order to improve the possibilities of automatic transformations UWE to ActionGUI, some extensions of the UWE Profile v2.1 were necessary, which are described in this section.

Content Model. UWE also uses the well-defined OCL from ActionGUI's SecureUML Model that refers to the user of an action in the web application as '`caller`'. Consequently, we have to specify the class representing the caller (i.e. the user), which is implemented by the stereotype ≪webUser≫.

Connection of Menus. All methods that are contained in a ≪navigationMenu≫ class have to be connected to the corresponding menus or anchors that are used in the UWE Presentation Model. This is done by adding a new stereotype ≪menuLink≫ with the tag {target : Anchor} to the `Operation` class, as depicted in figure 20.

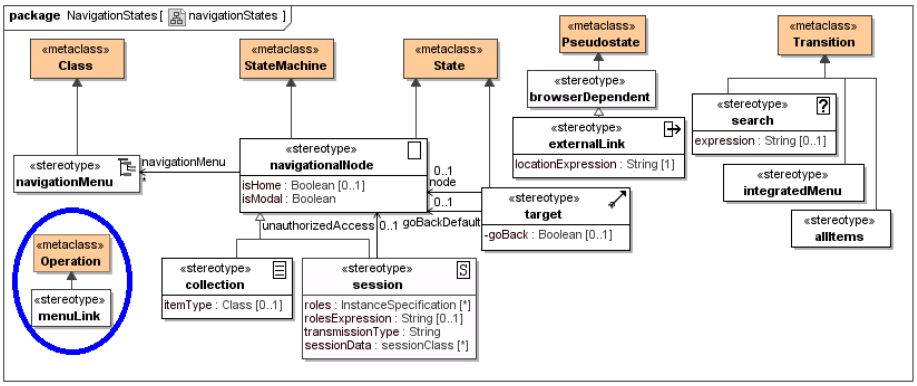


Figure 20: UWE navigational states profile

Navigational States Model. In order to identify in which situations the menu of a web application should be recalculated, the stereotype ≪updateMenus≫ is introduced which should be set manually on transitions in the UWE Navigational States Model (figure 20).

Presentation Model. The relationship between a navigational state (or its parental state) and a presentation group needs to be modeled explicitly in UWE in order to facilitate the mapping to ActionGUI. Therefore the tag {navState : State} has been added to ≪presentationGroup≫ elements of the UWE Presentation Model (figure 21).

Every transition of the UWE Navigational States Model is triggered by a certain element of the UWE Presentation Model. To establish a connection between those two elements, the tag {navTrans : Transition} has been introduced in ≪interactiveElement≫ s of the UWE Presentation Model.

For supporting menus explicitly in UWE, we have added the stereotype ≪presentationMenu≫ that inherits from ≪anchor≫. For compatibility with ActionGUI only ≪anchor≫ s should be added as property to a concrete menu class.

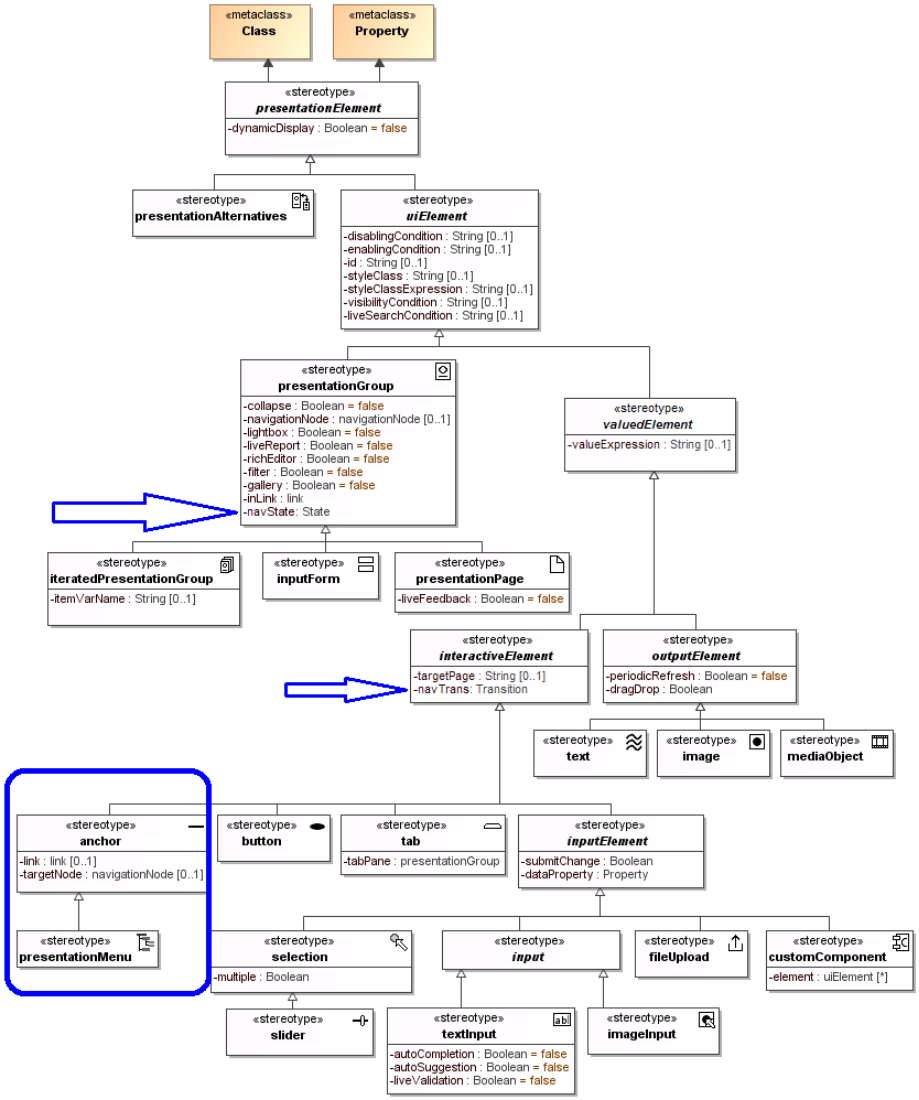


Figure 21: UWE presentation profile

7 Conclusion and Outlook

In this document we described *ActionUWE*, the transformation of UWE models into ActionGUI models. As ActionGUI models can be transformed to code, ActionUWE is an approach to semi-automatically generate secure software from UWE models.

Summarizing, one might say that *UWE* enables the modeler to start modeling on a high level of abstraction, to design different views and to use the well-known UML modeling language. ActionUWE can be used for the generation of concrete *ActionGUI* models. This is the time for the web engineer to be more specific about the behavior of the application in order to use the ActionGUI generator to produce an executable prototype.

The ActionUWE transformation itself is executed in four steps:

Step 1 Initializes the ActionGUI model and transforms available UWE menus to ActionGUI `Menu` classes.

Step 2 adds further information of the Presentation Model to ActionGUI.

Step 3 transforms the UWE Navigational States Model to ActionGUI Model without regarding security features.

Step 4 converts the role based access control (RBAC) constraints and the navigational access control features, modeled in the UWE Basic Rights Model and the UWE Navigational States Model.

To make the transformation possible, we had to slightly extend the metamodels of UWE and ActionGUI. Furthermore, we had to specify sound preconditions.

For the first approach of the transformations we e.g. assumed that one menu exists and this menu changes exactly one other panel. This is necessary to keep the planned implementation as simple as possible. In further versions this restriction might be avoided, but this would lead to a rather complex way of describing which panel (or subordinated panel) should be exchanged at runtime.

As illustrated in the *HospInfo* example, ActionGUI allows buttons as menu entries. This might be changed in a way that links/buttons and other elements like a text field for the search can be inserted, as common for huge web applications.

One resulting question for a future implementation is, whether ActionUWE should transform every piece of information from UWE that is expressible in ActionGUI (e.g. by the use of Choice statements) or whether security features of UWE's Navigation Model could be ignored for the transformation process. The former approach has been explained in this report. The latter would also be possible, because ActionGUI can infer some navigational access control rights from SecureUML Model and hide elements and pages with restricted access. Nevertheless, a model-checker would be required to double-check, if the user of the transformed application still has access to the same menus and pages (as sometimes a page should be removed from the menu when it contains a prohibited action and sometimes it should be accessible and just the element that triggers this action should be hidden). Presumably, a mix of both approaches should be implemented in the long run.

At the moment we are working on the implementation of the transformation, starting with a simple address book example without menus. After this proof-of-concept we plan to extend the tool support for both, ActionGUI and UWE to support e.g. semi-automatic tag-links between the several UWE models.

8 Acknowledgements

We would like to thank Martin Wirsing and Manuel Clavel for supervising this work. Furthermore, we are thankful for the NESSoS funding of the travels to Madrid and Munich and thus for the opportunity to work together face to face. Finally, our thanks go to Nora Koch for extensive proof reading.

References

[1] D. A. Basin, M. Clavel, M. Egea, M. A. G. de Dios, C. Dania, G. Ortiz, and J. Valdazo. Model-driven development of security-aware guis for data-centric applications. In A. Aldini and R. Gorrieri, editors, *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volume 6858 of *Lecture Notes in Computer Science*, pages 101–124. Springer, 2011.

[2] M. Busch. Integration of Security Aspects in Web Engineering. Master's thesis, Ludwig-Maximilians-Universität München, 2011. `http://uwe.pst.ifi.lmu.de/publications/BuschDA.pdf`.

[3] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. In J. Grabis and M. Kirikova, editors, *10th Int. Conf. on Business Perspectives in Informatics Research*, LNBIP, pages 239–253. Springer Verlag, 2011.

[4] N. Koch, A. Knapp, G. Zhang, and H. Baumeister. UML-based Web Engineering: An Approach based on Standards. In *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 157–191. Springer, 2008.

[5] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. 5th Int. Conf. Unified Modeling Language (UML'02)*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[6] S. Meliá, J. Gómez, S. Pérez, and O. Díaz. A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In *Proc. 8th Int. Conf. Web Engineering (ICWE'08)*, pages 13–23. IEEE, 2008.

[7] N. Moreno, P. Fraternali, and A. Vallecillo. WebML modelling in UML. *IET Software*, 1(3):67, 2007.

[8] F. Valverde and O. Pastor. Applying Interaction Patterns: Towards a Model-Driven Approach for Rich Internet Applications Development. In *Proc. 7th Int. Wsh. Web-Oriented Software Technologies (IWWOST'08)*, 2008.