Journal of Web Engineering, Vol. 15, No.1&2 (2016) 045-083

XANUI: A TEXTUAL PLATFORM-INDEPENDENT MODEL FOR RICH USER INTERFACES

JESÚS M. HERMIDA

European Commission, Joint Research Centre (JRC)

jesus.hermida@jrc.ec.europa.eu

SANTIAGO MELIÁ, ANTONIO ARIAS

Department of Software and Computing Systems, University of Alicante, Spain

{santi, aarias}@dlsi.ua.es

Received January 20, 2015
Revised September 22, 2015

In recent years, several model-driven proposals have defined user interface models that can represent both behavioural and aesthetic aspects. However, the software industry has ignored the majority of these proposals because the quality of the rich user interfaces generated out of these models is usually low and their code generators are not flexible, i.e., the UI templates cannot be customised easily. Furthermore, these proposals do not facilitate the separation between the visual design of the UI, normally performed by graphic designers in the industry, and the visualisation of data, which has been previously modelled using another domain-specific language.

This paper proposes a new textual domain-specific language called XANUI, which could be embedded in XML-based UI pages, e.g., HTML or XML. The designed language provides the mechanisms to bind visual components with data structures already existing, and to define the behaviour of these components based on events. In this paper, XANUI is integrated in two OOH4RIA development processes, i.e., the traditional data-intensive and the new design-first process, thus reusing the OOH4RIA models and transformations to generate a complete rich Internet application for any platform or device. In order to validate this approach, the XANUI solution is applied to the development of a RIA with two UI types: a) the administration view of a Web application using HTML5 and AngularJS, and b) a catalogue application for e-Commerce using Windows RT in a Tablet PC.

Key words: XANUI, OOH4RIA, User Interface Design, Model-Driven Development, Web Engineering
Communicated by: J.C. Preciado, F. Sánchez-Figueroa, & G. Rossi

1 Introduction

In modern software applications, the user interface is considered as the most costly layer in terms of design, development and maintenance due to three reasons: a) the large number of languages available (HTML5, CSS, JavaScript, etc.); b) designers need good knowledge of human-computer interaction (HCI) techniques, either regarding the aesthetic features or the data organisation in the UI; and c) the communication between this application layer and the others, which could be asynchronous and distributed.

During the last decade, reducing the time and effort of development of rich user interfaces has been a recurrent problem, tackled by some model-driven proposals in the fields of Web Engineering (e.g., WebML [4], RUX-Method [12], OOH4RIA [13], etc.) and Human-Computer Interaction (e.g., USIXML [11], TERESA [2], MB-UID [3], etc.) These proposals define a set of models that can represent the aesthetic and behavioural aspects of the user interfaces with the aim of improving the performance of the UI development and the maintenance tasks, as well as reducing the number of errors in the UI.

Despite their potential benefits, the majority of these approaches have not been adopted by the industry due to: 1) the quality of the design of the rich user interfaces generated from the proposed models is usually lower than the one resulting from the industrial UI design tools; 2) their code generators are not flexible, i.e., the templates cannot be customised easily; and 3) these proposals do not facilitate the separation of the work of graphic designers, normally in charge of the visual design of the rich UI, and other software developers/engineers, who develop how data is presented to the users using other domain-specific languages.

In addition, technologies for rich UIs evolve faster than these model-driven approaches, whose development cycles are longer. Model-driven approaches need to adapt to the changes in the modern technologies for UI development in different aspects, such as data binding, event-driven interactions between widgets or asynchronous invocations. Moreover, they need to support the generation of code following the good practices of the UI architectural patterns (e.g., Model-View-Viewmodel or Model-View-Presenter). The cost of these adaptations slow down the development process of the model-driven approaches, which are always one step behind the UI frameworks/technologies.

In this context, this paper introduces a new domain-specific language called XANUI (eXtensible ANnotation-based User Interface) that addresses the problems of the traditional model-driven approaches for rich UI development. XANUI defines a new textual language based on a set of embeddable annotations that allows designers to represent the behaviour of modern user interfaces independently from the UI framework in which the UI static components are coded, e.g., HTML or XML. Thus, this DSL facilitates that graphical designers define their UI structural views using professional design tools (e.g., Dreamweaver, Expression Blend, Android Studio, etc.) while software developers can define the behaviour of these views by means of several mechanisms, e.g., data binding, events, etc. This approach follows the principles of the Model-View-Controller pattern applied to UI models, thus facilitating the collaboration between both actors, i.e., graphical designers and UI software developers, and the synchronization between the user interface designs and the generated code.

In order to assess the features proposed, XANUI was integrated as a frond-end model in the OOH4RIA development process [13, 14]. In this way, this paper also proposes a full-stack model-driven solution for the development of Rich Internet Applications (RIAs) based on three models: a) a domain model for the definition of the data persistence and the business logic, b) the service model for the specification of the application façade (based on SOAP or REST), and c) the new XANUI model for the design of rich user interfaces. In OOH4RIA, XANUI can be used in two alternative development processes: 1) a bottom-up approach, starting from the modelling of the domain as in the traditional processes of Web Engineering; or 2) a design-first approach, starting from the design of a set of user interface templates.

This solution is supported by the OIDE tool (OOH4RIA Integrated Development Environment, *http://suma2.dlsi.ua.es/ooh4ria/*), based on the Eclipse Modelling Framework. More specifically, the XANUI editor was implemented using the Xtext framework (*http://www.xtext.org*). This editor provides a large number of benefits for designers, such as compilation, syntax highlighting, syntax validation and code generation, thus facilitating the adoption and use of XANUI by designers or developers.

The complete approach is validated by means of a case study, i.e., the development of a RIA for shop catalogues called inaCatalog. The application has two types of rich UIs with different technologies and features. The first case study consists in the development of a backend data-intensive RIA, whose UI was designed in HTML5 and the responsive Bootstrap framework, following the bottom-up OOH4RIA process. The second case study consists in the development of a catalogue of the product offered by a set of suppliers using a tablet PC application. In this second case, the OOH4RIA design-first process was followed, using the Blend tool to design the UI in XAML.

The paper is organised as follows: Section 2 explains the new model-driven development process of OOH4RIA using an SPEM notation, including the XANUI DSL. Section 3 presents the main contribution of this work, i.e., the XANUI model, with its main features, elements and examples of use. Subsequently, Section 4 briefly describes the code generation processes from XANUI and the OOH4RIA models. Section 5 describes the case study developed to demonstrate the feasibility of this approach. Section 6 summarises the relevant solutions similar to XANUI while, in Section 7, the authors analyse the main benefits and issues of the proposal. Finally, Section 8 draws the main conclusions and briefly defines the future lines of work.

2 A Model-driven Development Process for Advanced User Interfaces

The majority of the Web engineering methodologies follows a data-intensive approach [7], which defines a single-way model-driven process that start from the domain representation, define the most relevant data paths in a service model, and conclude with the design of the user interface. As some authors claimed [18], these approaches usually fail in providing agile interaction with customers because the process is strict and specific results are obtained in late stages of the process. Other Web engineering approaches such as MockupDD [17] propose an alternative process, in which the designers start by gathering the interaction requirements by means of UI mock-ups. Subsequently, from these mock-ups, a presentation and a service model are generated. Finally, a basic conceptual model is inferred and further refined.

The XANUI proposal can be integrated in the OOH4RIA methodology, thus defining a two-way model-driven development process, i.e., with support to both process types and allowing designers to select the most appropriate for their particular Web application. Figure 1 and Figure 2 show two standard SPEM 2.0 diagrams that depict the updated versions of the OOH4RIA development process [13] with XANUI.

More specifically, Figure 1 shows the data-intensive OOH4RIA process with XANUI. This process starts with the specification of the OOH4RIA Domain model, which is a platform-independent model representing the data entities, with their attributes and operations, and their relationships (inherited from the OO-H methodology [10]). Subsequently, the next activity is to define the Service

model (formerly known as Navigation model), in which designers specify the service interface and the data transfer objects, exchanged between the RIA server and the client. These models define the most relevant paths through the information space, i.e., they filter the data objects that will be visualised in the client side and the operations allowed to the users. From these two models, the RIA server side, i.e., the business logic and the service façade components, can be generated by means of a set of model-to-text transformations.

From the Service model, a set of user interface templates can be generated using the XANUI language and the technology chosen by the designer. This transformation provides a data-intensive user interface that could be used as a basic view of the backend data. Graphical designers can modify the design of the templates with no side effects on the synchronization with the server models. The result of the transformation can be used as a learning example by developers with little experience in the language.

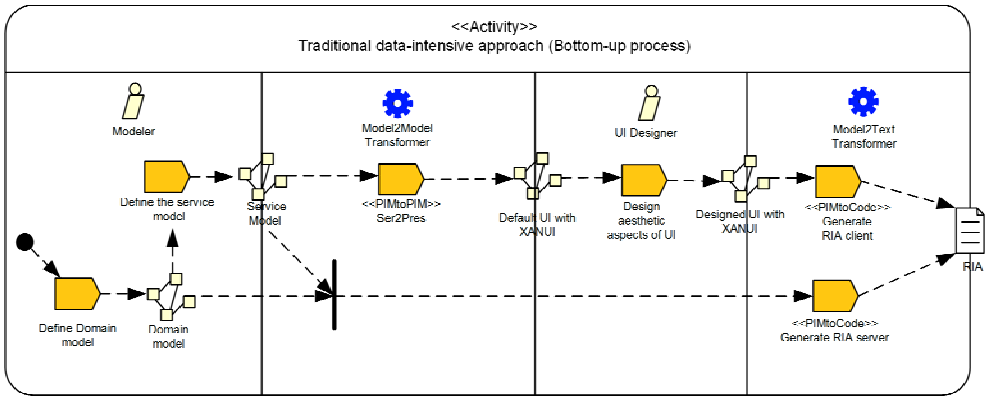

Figure 1. SPEM2 diagram of the traditional OOH4RIA data-intensive development process using XANUI.

Figure 2 illustrates the new OOH4RIA design-first model-driven process, which is focused on the specification of the UI from early phases of development. The process starts with the design of the UI, in which a UI Designer defines a set of low-fidelity UI prototypes to gather early feedback from the end users. Once the end-user approved these mock-ups, they can be further developed and transformed into high-fidelity prototypes or static UI templates using industrial UI design tools (Expression Blend, Flash, Dreamweaver, etc.). In the UI prototypes, designers specify a set of the static UI templates that show the structure of the view and the aesthetics aspects (e.g., the layout size, the screen dimensions, the spatial position of widgets and the style). The static UI templates correspond to the view component of a RIA client.

Designers could potentially choose any UI technology as long as they apply the Model-View-Controller pattern [6] and a declarative language, e.g., Microsoft .NET's XAML, HTML5 for Web applications, XML for Android, XML for iOS, etc. This choice is also limited by the number of code generators available for XANUI, which are specific for each UI technology.

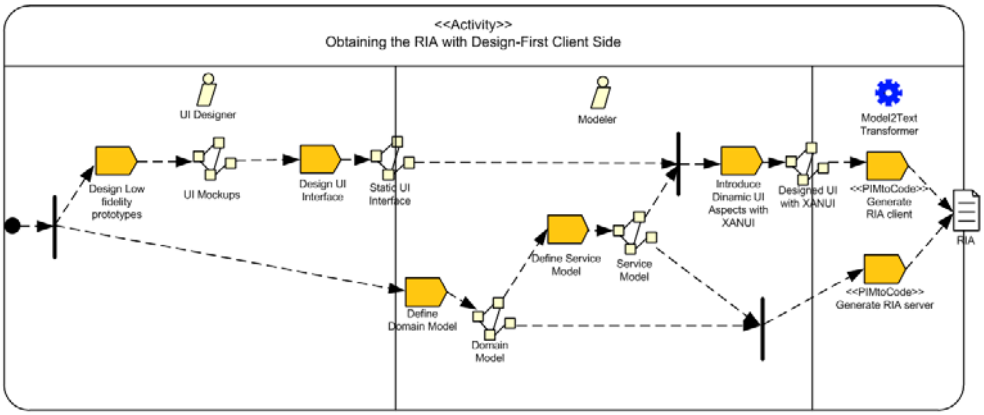

Figure 2. SPEM2 diagram of the design-first OOH4RIA development process with XANUI.

At the same time, the modeller can specify the RIA server using the Domain and Service models. Thus, this process facilitates and encourages the work in parallel of the two experts in RIA development: the designer, with the User interface; and the modeller or developer, with the application server

Once the Service model and UI templates are defined, the modeller can insert the XANUI model annotations into the designed templates in order to define the UI behaviour and to connect them to the services defined in the server side. In the final step, the final RIA implementation is generated through the “Generate RIA client” and “Generate RIA server” model-to-text transformations.

3 XANUI

XANUI is a domain-specific language with a textual notation that facilitates the specification of the dynamic components of advanced user interfaces, e.g., bindings between the data elements managed by the server and the UI components; or the management of UI events, needed to define the interaction between the UI components or the invocation of services from the server components. XANUI reuses and extends the design primitives originally provided by the OOH4RIA Presentation and Orchestration models.

XANUI can be used independently from the UI technology/framework and the service interface, i.e., it is a Platform-Independent model. It provides a set of textual annotations that can be embedded in the UI code in order to represent the structural elements as well as their behaviour. These annotations do not modify the default organisation of the view nor the aesthetic aspects.

XANUI does not define data services but reuses their specification from other models. In this paper, the service definitions were reused from the OOH4RIA Service model. As mentioned in the last Section, with this OOH4RIA model, designers can specify the service façade of the application through the definition of a set of service classes, i.e., Data Transfer Objects (DTO), and the operations allowed on them (CRUD operations or custom operations). XANUI only imports the definition of the service façade and specifies how the services will be used by the RIA client.

The following section will introduce the analysis of the UI technologies performed in order to define the main XANUI features. Subsequently, the paper will describe the main features of this

language, i.e., its abstract and concrete syntaxes as well as a set of usage examples. More specifically, the next subsection will introduce the generic concepts used in XANUI to represent the UI in a manner independent from the XANUI concrete syntax. This will facilitate the understanding of the modelling process with XANUI as well as the decisions taken in the design of the abstract and concrete syntaxes.

3.1 Analysis of the Existing Frameworks for UI Development

Before the design XANUI, the most relevant technologies/frameworks for UIs in Rich Internet Applications, desktop applications and mobile applications were studied 1) to assess the feasibility of the XANUI approach, 2) to analyse which UI primitives from OOH4RIA could be reused or needed an update; 3) to identify new design primitives not covered by OOH4RIA. Table 1 shows the UI technologies analysed and their programming languages for coding the view (or the static components of the UI) and the controller (i.e., the behaviour of the UI).

Table 1. Analysis of the current UI frameworks.

Platform	User Interface Frameworks	Programming Languages	
		View	Controller
Desktop	**Windows Presentation Foundation (WPF)**	XML (XAML)	C#
	WindowsForm	C#	C#
Mobile	**Android**	XML	Java
	iOS (Xamarin)	XML (XIB)	C#
	Windows RT	XML (XAML)	C#
RIA	**Adobe Flex**	Binary	ActionScript
	AngularJS	(X)HTML, CSS	JavaScript
	React	(X)HTML, CSS	JavaScript
	Backbone.js	(X)HTML, CSS	JavaScript
	Silverlight	XML (XAML)	C#
RIA/Desktop	**JavaFX**	XML (FXML)	Java

The application view in browser-oriented RIAs is frequently represented using HTML and CSS (AngularJS, React) and, in plugin-oriented ones, using XML (Silverlight and JavaFX). In desktop technologies (WPF and JavaFX), views can also be represented using XML (an exception is the WindowsForm platform, which represents the view and the controller using C#). Regarding the mobile frameworks, the most relevant mobile platforms in terms of number of terminals, i.e., iOS (Xamarin), Android and WindowsRT, represent the view by means of XML-based languages.

The majority of the technologies analysed use a declarative specification of the view based on XML-based languages (e.g., XHTML or XAML). XANUI should therefore be adapted to the design of this type of views.

For the application controller, each technology uses a different language (C#, JavaScript, ActionScript) depending on the platform and the type of application. For this reason, XANUI should use a generic language for the UI behaviour, which could be translated into any of the existing.

The study also identified which OOH4RIA primitives could be reused/adapted in XANUI and which new ones should be added. The original OOH4RIA UI primitives were inspired by HCI multiplatform languages, such as USIXML [11] or UIML [1], and the industrial Rich Internet Applications toolkits available in 2008, such as GWT, Silverlight, Flex, etc. From the conception of OOH4RIA, UI technologies have been rapidly developed including new features and styles. Therefore, a new review of the technologies was needed to ensure the relevance of the design primitives included in XANUI. Table 2 summarises the results of the second and third analysis.

Table 2. Analysis of the design primitives for the current UI frameworks/technologies.

Reusable primitives from OOH4RIA	**Required Primitives (not in OOH4RIA)**
• Widget definitions (platform-independent) • Definition of data contexts • Event-Condition-Action rules • Event definitions (predefined for each widget) • UI actions (predefined for each widget)	• Templating (definition and instantiation of model templates) • Data binding (link between the UI model and the server data) • Definition of data scopes • Validation rules for UI widgets • Custom event definitions • Custom UI action definitions

3.2 Representing the User Interface with XANUI

XANUI can represent two types of UI elements: structural, i.e., related to the structure of the UI; or behavioural, which specify the behaviour of the interface according to the interaction with the users and the server services.

The basic structural elements of the model are the following:

a) Widgets, which are the basic components in the user interface. There are two types of widgets: widgets for data representation, used to visualise data (e.g., a label), and interaction widgets, which enables the interaction between the UI and the users.

b) Containers, which are groups of widgets (and/or containers) arranged according to a disposition order (e.g., a stack panel or a canvas). The relationship of containment is transitive.

c) Models, which are representations of the complete view that is shown to the users.

Structural elements can be related to the service definitions from the OOH4RIA Service model by means of three relationship types (illustrated in Figure 3):

a) Context. The context of a model, a container or region of the model defines the type of data object (or service class) that can be visualised in their area and the service operations that can be invoked. A model can only be associated to a single data object while a container can also visualise a collection of objects. All models must have a context.

b) Navigation. This relationship specifies a subcontext of a previously defined context. The new subcontext must be linked to the parent context by means of service operation from the Service Model. The parent context will be the input of the operation and the result will be shown in the subcontext.

c) Binding. It relates a widget to an attribute of any of the contexts in which it is contained. In this way, a widget can visualise data from the context or can store data in the context. Widgets can be associated to any of the contexts in which they are contained.

Figure 3 illustrates the structural elements represented in XANUI and their relationships with the OOH4RIA Service Model. For each UI view, XANUI defines an associated model using the corresponding annotations. Within the model, the designer can annotate the relevant containers, which could be nested, and the widgets contained. In the figure, there are two views: one for anonymous users and another for registered users.

Once the relevant structural elements are annotated, designers should define their relationships with the OOH4RIA Service model. In the example, the first model is related to the *AnonymousUser* service class offering the operation "login". The second model is related to the *User* service class (i.e., registered user). The outer container is linked to the *Record* service class through the *getAllAlbum* service operation, thus creating a subcontext. The widgets in the example are associated to different attributes. Depending on the binding type and the widget type, widgets will visualise the data or will be also capable of saving the user input into the context.

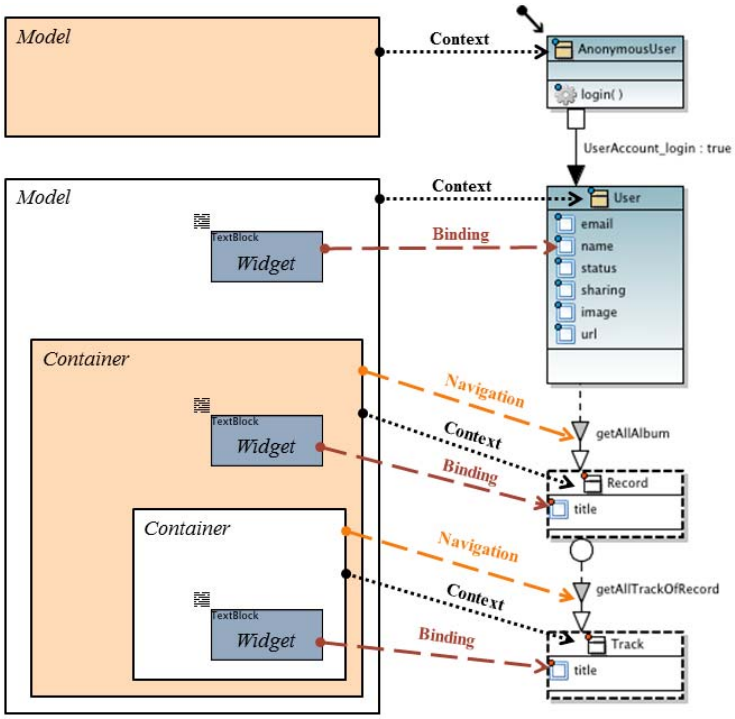

Figure 3.Example of the relationships between XANUI and the OOH4RIA Service model.

XANUI defines the UI behaviour as a set of ECA rules (Event, Condition, Action), i.e., these rules can specify which actions the UI (or the application) should perform after users trigger events on the UI structural components. In ECA rules, events represent the interaction between users and widgets

(e.g., clicking a button); conditions control the flow of actions performed after an event was triggered; and the actions can be carried out by the UI or by the server components.

For each widget, there is a set of events that a user can trigger another set of (client) actions that it can perform, which mainly depend on the technology of the implementation. For each event that might be triggered, the designer defines a set of conditions that control the behaviour of the application. If a condition is satisfied, the set of actions associated to the condition is performed. New conditions over the action results can be also specified, thus creating a chain of actions.

The concept of ECA rule was reused from the OOH4RIA Orchestration Model [15]. Figure 4 shows an illustrative representation of an ECA rule, in which the designer defined the behaviour of a button after a user clicked on it.

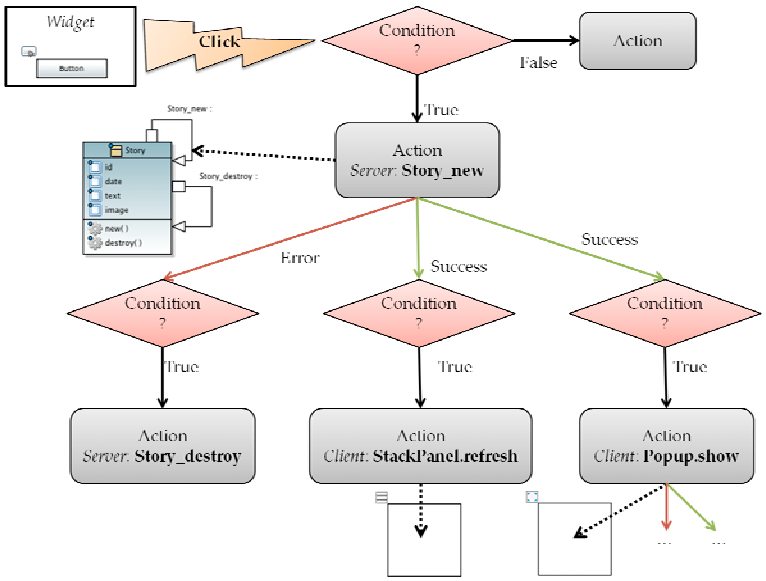

Figure 4. Representation of an Event-Condition-Action rules.

3.3 Abstract Syntax

The XANUI metamodel captures the information of the main elements of the language. The metamodel elements can be classified into two groups: a) structural elements, i.e., elements representing visual components of the user interface and their relationship with the data elements from the OOH4RIA Service model; and b) behavioural elements, i.e., elements that represent the interaction between the users and the structural elements.

The diagram depicted in Figure 5 shows the structural elements of the XANUI metamodel, based on the ECORE meta-metamodel. The root of a XANUI model is the PresentationModel element with a VisualBlock element which can contain a set of expressions: a) Literal elements, i.e., the code of the user interface; b) RichString elements, i.e., annotations; c) other VisualBlock elements, defined in new UI contexts. There are six main types of annotations (i.e., RichString elements):

- Widget: basic structural element of the user interface. They can be simple (i.e., Widget element from Section 3.2; metaclass SimpleWidget) or a container (metaclass Container, Container element from Section 3.2).
- Context: Region of the model in which data objects will be visualised (linked to a ServiceClass and a ServiceLink element, from the OOH4RIA Service Model).
- Binding: Link between the data objects of the context and the Presentation model or the widget.
- Include: Instantiation of a model template.
- Behavioural annotations (figures 6 and 7): EventCall, CustomEventCall, EventDefinition and ActionDefinition.
- Validation annotations, which define rules to validate the content of the widgets (Constraint metaclass, single rule; ConstraintGroup, group of rules).

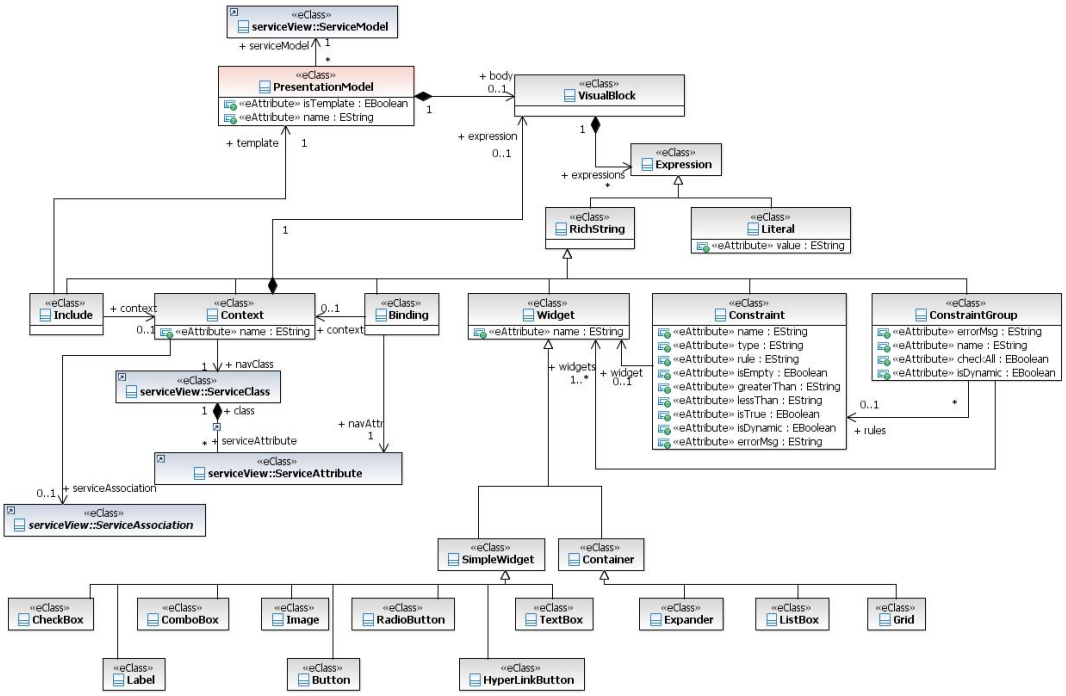

Figure 5. XANUI metamodel for UI structural elements and data contexts.

Figure 6 shows the meta-elements representing the ECA rules (from the *EventCall* metaclass) and the relationships to the structural elements and the elements of the Service model.

The main elements of an ECA rule in XANUI are the following:

- EventCall: Root of the ECA rule. It represents the event triggered.
- Condition: It manages if an action will be performed.

- ActionCall: Invocation to an action.
- Action. Possible actions that can be performed. There are three types of actions: a) ServerAction, e.g., the invocation of a service operation from the Service Model; b) ClientAction, e.g., a change in the properties of a widget; c) ContextAction, e.g., a change in the data objects associated to a model region (Context metaclass); and d) ScopeAction, e.g., a modification in the scope data objects.
- Argument. These elements assign a value to the arguments of the actions (ActionArgument). The value of an action argument can be a string value, the result of a previous operation (ActionResult) or bound to widget property (ActionArgumentViewBinding) or from the context (ActionArgumentModelBinding).
- ActionResult. They define a variable in which the result of an ActionCall is stored.

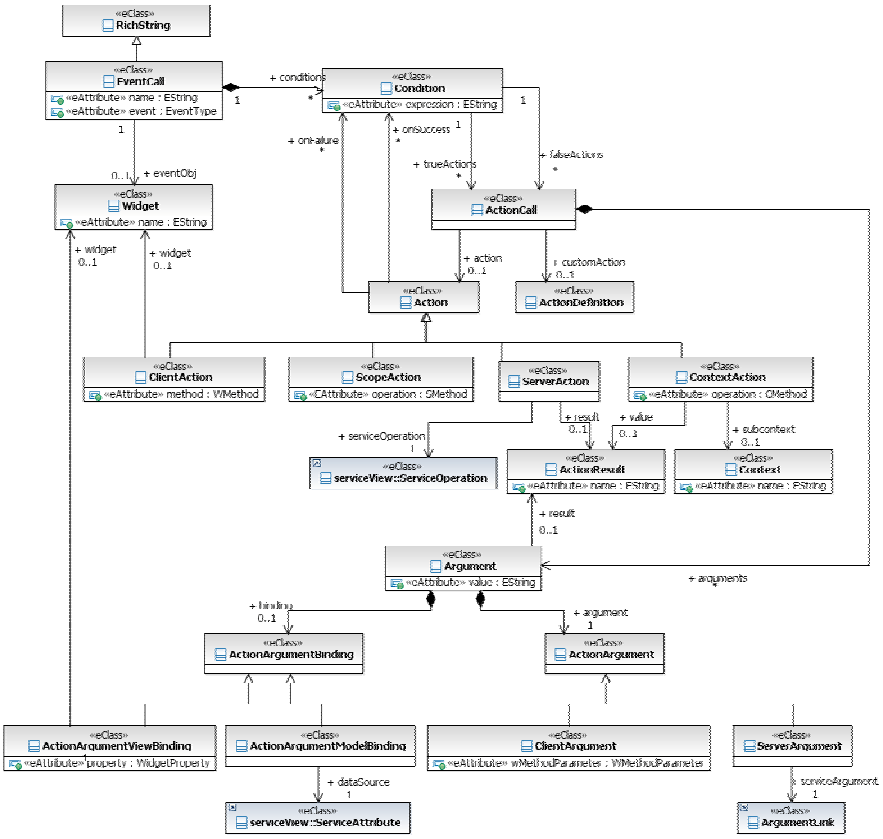

Figure 6. XANUI metamodel for Event-Condition-Action rules.

Figure 7 depicts the XANUI elements that can be used to extend the ECA rules.

- EventDefinition: Definition of a custom event.

- ActionDefinition: Definition of a custom action, in which the designer can include a piece of code. Designers can define the parameters of a custom action (Parameter).
- CustomEventCall: Definition of a custom ECA, in which the designer can include a piece of code.

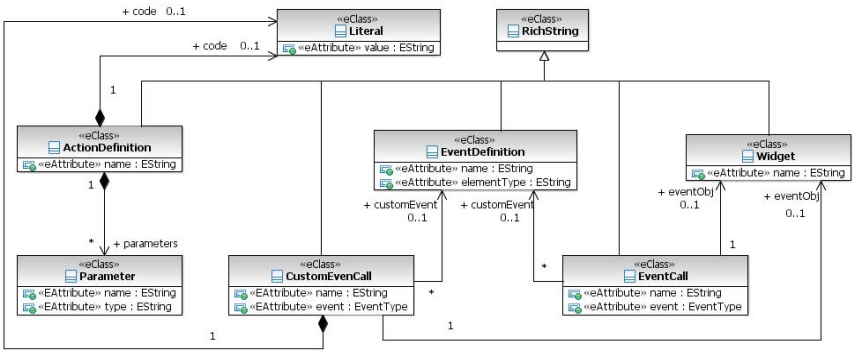

Figure 7. XANUI metamodel for custom behavioural elements.

3.4 Concrete Syntax

The concrete syntax for XANUI is specified by means of a grammar, which defines the textual representation of the elements of the XANUI abstract syntax. Figure 8 shows the main rules of the XANUI grammar using the Xtext syntax. A more detailed description of the grammar will be introduced in Section 3.5.

```
terminal TEXT : '%' '-' '-' '>' (!OPENING)* (EOF|OPENING);
terminal OPENING : '<' '!' '-' '-' '%';

/* Grammar rules */

PresentationModel:
        OPENING 'CONTEXT' (isTemplate?='TEMPLATE')? name=Fqn '@' context=[ServiceClass|Fqn]
        body=RichString ;

RichString:
        expressions+=RichStringLiteral
        (expressions+=RichStringPart expressions+=RichStringLiteral)* ;

RichStringLiteral:
        value=TEXT ;

RichStringPart:
        Context | Include | Binding | Widget | Eca | CustomEca | NewEvent | NewAction ;

Context:
        'SUBCONTEXT' name=Fqn
        ('@' navigation=[ServiceAssociation|Fqn] )?
        ('>>' class=[ServiceClass|Fqn] )?
        expression=RichString
        'ENDSUBCONTEXT' ;

Widget:
        SimpleWidget | Container ;

SimpleWidget:
        Button | HyperlinkButton | Label | Checkbox | RadioButton | Textbox | ComboBox | Image;

Button:
        "BUTTON" name=EString ;
```

```
/* ... */

Container:
        Grid
        | ListBox
        | Accordion ;

ListBox:
        "LISTBOX" name=EString
        ('@' navigation=[ServiceAssociation|Fqn] )?
        ('>>' class=[ServiceClass|Fqn] )?
                expression=RichString
        "ENDLISTBOX" ;

/* ... */

/*
 * Rules for ECA definition
 */

Eca returns EventCall:
        {EventCall}
        "ECA" name=EString
        '['
        (eventObj=[Widget|Fqn] | 'CONTEXT') '::' (event=EventType | customEvent=[NewEvent])
        ']'
        (conditions+=Condition)+ ;

Condition:
        (trueActions+=ActionCall)+
        |
        ('if' '(' expression=(EString|'true'|'false') ')'
        '{'
                (trueActions+=ActionCall)+
        '}'
        ('else'
        '{'
                (falseActions+=ActionCall)+
        '}')?) ;

ActionCall:
        action=Action '(' ( arguments+=Argument ( ',' arguments+=Argument )* )? ')'
        (result=ActionResult)?
        ('OnSuccess'
        '{'
                ( onSuccessConditions+=Condition )+
        '}')?
        ('OnFailure'
        '{'
                ( onFailureConditions+=Condition )+
        '}')?
        ';' ;

ActionResult:
        'AS' name=EString       ;

Action:
        ServerAction | ClientAction | ContextAction | ScopeAction ;

ServerAction:
        serviceAssociation=[ServiceAssociation|Fqn] ;

ClientAction:
        widget=[Widget|Fqn] '->' method=WMethod ;

ContextAction:
        ('CONTEXT' | subcontext=[SubContext|Fqn]) '->' operation=CMethod ;

ScopeAction:
        ('SESSION' | 'APP' | 'PAGE') '->' operation=SMethod ;

Argument:
        argument=ActionArgument
        '='
        (binding=ActionArgumentBinding
                | 'val' value=EString
                | 'res' result=[ActionResult|Fqn]);

ActionArgument:
        ServerArgument | ClientArgument ;

ServerArgument:
        argumentLink=[ArgumentLink|Fqn] ;
```

```
ClientArgument:
        wMethodParameter=WMethodParameter ;

ActionArgumentBinding:
        ActionArgumentModelBinding | ActionArgumentViewBinding ;

ActionArgumentModelBinding:
        dataOrigin=[ServiceAttribute|Fqn] ;

ActionArgumentViewBinding:
        widget=[Widget] '->' property=WidgetProperty ;

/* Validation rules */
Check:
        'CHECK' widget=[Widget]
                'TYPE' type=CheckType
                ('RULE' regex=EString |
                        ((isNotEmpty?='NOTEMPTY')?
                        ('GT' gt=EString)?
                        ('LT' lt=EString)?
                        (isTrue?='TRUE')?
                        (isFalse?='FALSE')?))
                (isDynamic?='DYNAMIC' | isStatic?='STATIC')
                'ERRORMSG' error=EString
                'AS' name=EString ;

CheckGroup:
        'CHECKGROUP' (rules+=[Check])+
                'APPLYON' (widgets+=[Widget])+
                (isDynamic?='DYNAMIC' | isStatic?='STATIC')?
                (checkAll?='CHECKALL')?
                ('ERRORMSG' error=EString)?
                'AS' name=EString ;

/* ECA extension rules */

NewEvent returns EventDefinition:
        'NEWEVENT' name=EString 'FOR' widgetName=EString ;

NewAction returns ActionDefinition:
        'NEWACTION' name=EString 'ON' ('MODEL' | widgetName=EString) (':' parameters+=Parameter)?
        code=RichStringLiteral
        'ENDNEWACTION' ;

Parameter:
        name=EString ( '(' type=EString ')' )? ;

CustomEca returns CustomEventCall:
        {CustomEventCall}
        "CUSTOMECA" name=EString
        '['
        (eventObj=[Widget|Fqn] | 'CONTEXT') '::' (event=EventType | customEvent=[NewEvent])
        ']'
        code=RichStringLiteral
        'ENDCUSTOMECA' ;

/* Rules for Fully Qualified Names (used to establish references between the model elements) */

FqnWithWildcard returns ecore::EString:
        Fqn '.*'?;

Fqn returns ecore::EString:
        EString ( '.' EString )*;

/* Basic types */

EString returns ecore::EString:
        STRING | ID;

EInt returns ecore::EInt:
        '-'? INT;
```

Figure 8. Main rules of the XANUI grammar using the Xtext syntax.

XANUI is a language based on annotations that can be embedded in the code of the UI interfaces. Several modern technologies for Rich User Interfaces define the UI views using XML languages (see Table 1). Therefore, the XANUI annotations are included in a custom XML comment clause: `<!--% XANUI annotation %-->`.

The following example (Figure 9) shows a XANUI model enriching a HTML web page. The main context of this web page is an object of the User class (line 1). The Web page will show the roles of the user within the H2 HTML tag. In addition, an HTML button is annotated with XANUI, which, for instance, enables the definition of ECA rules over events of this button. In this example, the ECA rule *Refresh* (line 15) is triggered after a user clicks on the button. This ECA rule retrieves the roles of the user defined as context (through the *getAllRole* method) and updates the content of the page.

```
1   <!--%CONTEXT main @User%-->
2   <html>
3           <body>
4           <!--%SUBCONTEXT roles1 >>getAllRole%-->
5           <div>
6
7           <h2><!--%LABEL roleNameLbl @Role.name FROM roles1%--></h2>
8
9           </div>
10          <!--%ENDSUBCONTEXT%-->
11
12          <button><!--%BUTTON refreshRoles%-->%--> Refresh roles </button>
13          </body>
14  </html>
15  <!--%ECA Refresh [showRoles :: OnClick]
16          getAllRole() AS result
17          OnSuccess
18          {
19                  roles1->SetContentData(p_obj = res result);
20          }
21  }%-->
```

Figure 9. Example of XANUI combined with XHTML code.

3.5 XANUI in Action

Based on the metamodel and the grammar, this section describes the main features of the XANUI language using illustrative examples. The main features of XANUI are: a) definition of data contexts; b) widget annotations; c) definition of data bindings; d) use of templates; e) event-driven behaviour; f) definition of scopes (session, page, application); and g) validation of the fields. Each of the following subsections will show one of the mentioned features.

3.5.1 Scenario

The XANUI examples introduced in this section assume that the server designer already specified the OOH4RIA Domain and the Service models of the final Web application, which, in this case, is a small social network site (Figure 10). The application will manage information from the users of the social network, their role and the stories they publish.

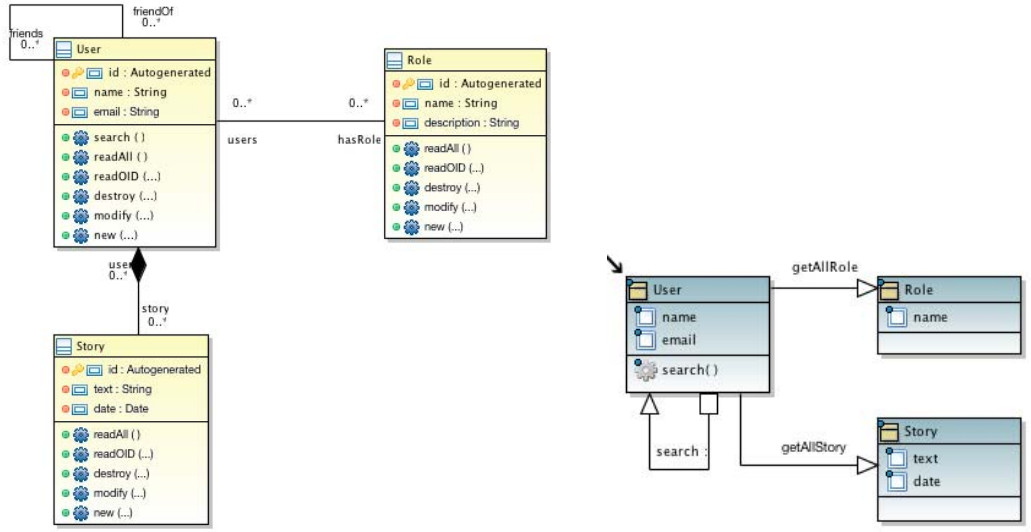

Figure 10. OOH4RIA Domain and Service models for the example.

3.5.2 Data Contexts

A UI document becomes a XANUI model when the designer defines the main data context using the elements of an existing OOH4RIA Service model. The context definition is the first annotation of a XANUI model and links the model to a data transfer object (i.e., a service class). Once the initial context is defined, the designer can bind the data of the context attributes (i.e., service attributes) to the visual representation.

XANUI models can define several subcontexts of the initial context on different regions of the UI. However, there is only one main context. Subcontexts can be defined from a) a service class, which they are linked to, or b) a service association, whose origin should be one of the parent contexts. The main difference between both cases is the process of loading data and visualise it within the context.

In the first case (see Figure 11, line 8, subcontext *friendInfo*), the service class only indicates which type of data can be visualised within the contents. In order to load and visualise the data from the server services, designers need to explicitly define an ECA rule (see Section 3.5.6) that specifies how to do it (e.g., which service will be invoked, the parameters and the subsequent UI behaviour).

In the second one (see Figure 11, line 5, subcontext *roles*), the service class related to the context is the destination of the service association (i.e., Role). By using a service association, the UI will automatically launch the process of retrieving the data from the server services and showing it to the end users while the page is loaded. The designer cannot modify the default behaviour.

Figure 11 shows the XANUI syntax for the annotations of the model element and the creation of subcontexts, together with an illustrative example of use.

```
XANUI Syntax:
        (Model context) context<!--%CONTEXT [MODEL_NAME] @[SERVICE_CLASS] (TEMPLATE)?%-->
        (Subcontext)    <!--%SUBCONTEXT [CONTEXT_NAME] @[SERVICE_CLASS]
                                (>>[SERVICE_ASSOCIATION])? %--> ... <!--%ENDSUBCONTEXT%-->

Example:
1       <!--%CONTEXT main @User%-->
2       <html>
3               <body>
4               <h2></h2>
5               <!--%SUBCONTEXT roles >>roles%-->
6               <h2></h2>
7               <!--%ENDSUBCONTEXT%-->
8               <!--%SUBCONTEXT friendInfo @User%-->
9               <h3></h3>
10              <h3></h3>
11              <!--%ENDSUBCONTEXT%-->
12              </body>
13      </html>

Notes:
        Before using an existing service model, it is necessary to import it with the sentence
        "IMPORTS [SERVICE_MODEL.*]".
        The CONTEXT and IMPORTS elements have no closing tag.
```

Figure 11. Definition of data contexts using XANUI.

3.5.3 Widget Annotations

Widget elements (or annotations) in XANUI establish a link between the XANUI model and the representation of the widget in the final software code. XANUI widgets will not generate new UI widgets from the annotations, but the widget must exist beforehand. Designers can use the widget annotations to define the behaviour of the widgets by means of ECA rules.

XANUI defines a taxonomy of widgets, either simple or containers, frequently used in most of the technologies. Designers should adequately annotate the widgets in their UI code with the XANUI annotations. By annotating the containers, the designers can also define a new subcontext in the same manner seen in Section 3.5.2.

```
XANUI Syntax:
        (Button)                <!--%BUTTON [WIDGET_NAME]%-->
        (HyperLinkButton)       <!--%HYPERLINKBUTTON [WIDGET_NAME]%-->
        (Textbox)               <!--%TEXTBOX [WIDGET_NAME]%-->
        (Checkbox)      <!--%CHECKBOX [WIDGET_NAME]%-->
        (RadioButton)   <!--%RADIOBUTTON [WIDGET_NAME]%-->
        (Label)         <!--%LABEL [WIDGET_NAME]%-->
        (Image)         <!--%IMAGE [WIDGET_NAME]%-->
        (Combobox)              <!--%COMBOBOX [WIDGET_NAME]%-->
        (Expander)              <!--%EXPANDER [WIDGET_NAME] @[SERVICE_CLASS]
                                (>>[SERVICE_ASSOCIATION])?%--> <!--%ENDEXPANDER%-->
        (ListBox)               <!--%LISTBOX [WIDGET_NAME] @[SERVICE_CLASS]
                                (>>[SERVICE_ASSOCIATION])?%--> <!--%ENDLISTBOX%-->
        (Grid)          <!--%GRID [WIDGET_NAME] @[SERVICE_CLASS]
                        (>>[SERVICE_ASSOCIATION])?%--> <!--%ENDGRID%-->
Example:
1       <!--%CONTEXT main @User%-->
2       <html>
3               <body>
4               <button><!--%BUTTON showRoles%--> Show roles </button>
5               <table><!--%LISTBOX friendList @User >>getFriends%-->
6                       <tr>
7                               <td><!--% LABEL nameLabel%--></td>
8                               <td><!--% LABEL emailLabel%--></td>
9                       </tr><!--%ENDLISTBOX%-->
10              </table>
11              </body>
12      </html>
```

Figure 12. Definition of widget references using XANUI.

3.5.4 Data Bindings

Data bindings establish a relationship between a widget and a property, i.e., a service attribute, of any of the subcontexts defined, including the main context (see Figure 13, line 4). This relationship is specified in the widget annotations. Binding can be one-way (i.e., read-only), two-way (i.e, read-write) or one-time (i.e., read only once).

If two subcontexts share the same service class, the binding relationship will be attached to the inner one unless designers use the FROM clause (see Figure 13, lines 9-10). This clause allows designers to choose the context owning the data.

```
     XANUI Syntax:
             (Data binding) <!--%[WIDGET_TYPE] [WIDGET_NAME] @[SERVICE_ATTRIBUTE] (FROM [CONTEXT_NAME])?
                                                  (oneway|twoway|onetime)? %-->

     Example:
1            <!--%CONTEXT main @User%-->
2            <html>
3            <body>
4            <h2><!--%LABEL UserName1 @User.name%--></h2>
5            <!--%SUBCONTEXT roles >>roles%-->
6            <h2><!--%LABEL RoleName @Role.name%--></h2>
7            <!--%ENDSUBCONTEXT%-->
8            <!--%SUBCONTEXT friendInfo @User%-->
9            <h3><!--%LABEL UserName @User.name FROM main%--></h3>
10           <h3><!--%LABEL friendName @User.phone FROM friendInfo%--></h3>
11           <!--%ENDSUBCONTEXT%-->
12           </body>
13           </html>
```

Figure 13. Definition of data contexts using XANUI.

3.5.5 Templating

XANUI facilitates the creation of model templates, which are independent models that can be instantiated in several regions of other models (even in new templates). In order to create a new template the initial context should be defined as TEMPLATE.

```
     Example:
1            <!--% IMPORTS NavModel.* %-->
2
3            <!--%CONTEXT userInfoTemplate @User TEMPLATE%-->
4            <h2><!--%LABEL UserName @User.name%--></h2>
5            <h2><!--%LABEL UserPhone @User.phonephone%--></h2>
```

Figure 14. Definition of a template using XANUI.

A template can be instantiated from the main context, a subcontext or an ECA rule.

```
     XANUI Syntax:
             (Include)       <!--%INCLUDE [MODEL_NAME] FOR [SUBCONTEXT_NAME]|CONTEXT%-->

1    Example:
2            <!--%CONTEXT main @User%-->
3            <html>
4                    <body>
5                    <!--%INCLUDE userInfoTemplate FOR CONTEXT%-->
6                    </body>
             </html>
```

Figure 15. Instantiation of a template using XANUI.

3.5.6 Extensible Event-Driven Behaviour

In XANUI, the behaviour of the UI widgets is specified by a collection of ECA rules. ECA rules can be defined in the same model or in a separate one (recommended).

Designers will define the UI behaviour based on the events that the users can trigger on the widgets. For each event, there is a set of conditions that control the invocation of the actions:

- XANUI provides a predefined set of UI events, including the most used in the technologies analysed (Table 1). Each widget type can trigger a specific set of events, which might be shared with other widgets.
- Conditions are optional and can specified using a subset of the OCL over the elements of the XANUI models. Conditions will be translated into the final language of the user interface.
- ECA rules can invoke four types of methods (actions): a) server, i.e., an invocation of a server service, represented by a service association; b) client, i.e., a method controlling the appearance of the widget; c) context, i.e., methods manipulating the data contained in each context; or d) scope, i.e., methods manipulating the data contained in the application-level contexts, i.e., the session and the application data repositories (see Section 3.5.7). The parameters of these methods can be obtained from values contained in a widget or from the result of a previous method.

  Each widget type can invoke a set of client methods, not extensible by the designers, with a predefined signature. The choice on the methods was made after an analysis of the existing technologies.

  Context methods mainly manipulate the content of the different contexts, i.e., retrieve or assign a value to a context. If the data object of a context is modified, the data objects of the subcontexts are updated accordingly (for those subcontexts created from a service association). Scope methods will be introduced more in detail in the subsequent sections.

  The method arguments can be free values (starting with the *val* clause), or can be bound to the attribute of a context, to a property of a widget or to the result of other operation (starting with the *res* clause).

The example in Figure 16 defines the behaviour of the *showRoles* button when a user triggers the *OnClick* event, i.e., they click on it (line 1). The *getAllRole* server method is invoked and the result is saved in the *result* variable (line 2). Depending on the results of the method (i.e., if there was any error or not, lines 3 and 11), the designer can define new conditions and new method invocations (line 5).

```
XANUI Syntax:

        (ECA rule) <!--%ECA [ECA_NAME] \[ [WIDGET_NAME|CONTEXT_NAME] :: [EVENT_NAME] \] [CONDITION]+ %-->

        (Condition) if ( [CONDITION_CLAUSE] )
                { [METHOD_INVOCATION]+ }
                else
                { [METHOD_INVOCATION]+ }

        (Method invocation) ((CONTEXT|SESSION|APP|CONTEXT_NAME|WIDGET_NAME) ->)? [METHOD_NAME]
                                \( [METHOD_ARGUMENTARGUMENT]+ \) AS [RESULT_NAME]
                        (OnSuccess
                        {
                                [CONDITION]+ | [METHOD_INVOCATION]+
                        })?
                        (OnFailure
```

```
                        {
                                [CONDITION]+ | [METHOD_INVOCATION]+
                        })?

        (Method argument) [ARGUMENT_NAME] = (val [VALUE] | [WIDGET_NAME] -> [WIDGET_PROPERTY] | res
        [RESULT_NAME] | [ATTRIBUTE_NAME])

    Example:
1       <!--%ECA show [showRoles :: OnClick]
2       getAllRole() AS result
3       OnSuccess
4       {
5               if (true) /* Optional Condition */
6               {
7                       roles2->SetContentData(p_obj = res result);
8                       SESSION->SetContentData(key = userRoles, value = res result);
9               }
10      }
11      OnFailure
12      {
13              ...
14      };
15      }%-->
```

Figure 16. Definition of an Event-Condition-Action rule using XANUI.

The definition of the ECA rules in XANUI can be customised to the needs of the applications and the designers. XANUI offers three main extension mechanisms for ECA rules:

a) Custom events. XANUI offers a predefined set of events for each widget. Designers can define new events if needed. The new events will be used in the same manner as the existing ones. Events can only be defined on a type of structural element, not on a specific widget or container.

```
XANUI Syntax:
        (Custom event)  <!--%NEWEVENT [EVENT_NAME] FOR [WIDGET_TYPE|CONTAINER_TYPE]%-->

Example:
        <!--% NEWEVENT longTap FOR Button %-->
```

Figure 17. Definition of custom events with XANUI.

b) Custom UI (client) actions. In ECA rules, XANUI offers predefined actions for each structural element. If needed, designers can specify new actions on the widget or container types. This creates a protected region in which designers can include the code of the action.

```
XANUI Syntax:
        (Custom action) <!--%NEWACTION [ACTION_NAME] ON [WIDGET_TYPE|CONTAINER_TYPE|MODEL] : [PARAMETER]+
                        %-->
                        [CODE]
                        <!--%ENDNEWACTION%-->

Example:
        <!--%NEWACTION updateColors ON MODEL : colorScheme %-->
        Javascript code | .NET code | Java code
        <!--%ENDNEWACTION%-->

Note:
        Designers must not include the method declaration in the code region.
```

Figure 18. Definition of custom actions with XANUI.

c) Custom ECA rules. In the third extension case, designers can define their own custom ECA rule. Custom ECA rules create a protected region in which designers can code the behaviour of the UI after an event is triggered on a widget or container.

```
XANUI Syntax:
       (ECA rule)      <!--%CUSTOMECA [ECA_NAME] \[ [WIDGET_NAME|CONTEXT_NAME] :: [EVENT_NAME] \] %-->
                       [CODE]
                       <!--%ENDCUSTOMECA%-->

Example:
       <!--%CUSTOMECA updateColors [ showRoles :: OnClick ] %-->
       Javascript code | .NET code | Java code
       <!--%ENDCUSTOMECA%-->
```

Figure 19. Definition of custom ECA rules.

3.5.7 Data Scope (Page, Session, Application)

In the ECA rules of a XANUI model, designers can also manage data objects from three data scopes (apart from the data stored in the contexts):

- At page level. Designers can store or retrieve data objects from the PAGE element, which is unique to the model, i.e., the data stored in one model cannot be used in another model.
- At user level. Designers can store or retrieve data objects related to a specific user in the SESSION element. Users cannot share the same SESSION element.
- At application level. Designers can store or retrieve data objects related to the application using the APP element. The data objects stored in this element can be shared among the application users.

These three elements store entries [key, value] in which keys are strings and values can be objects of any datatype. Their content can be managed by means of two main methods: GetContentData (to retrieve a data object) and SetContentData (to store a data object).

Furthermore, designers can manage the *OnChange* event (showing a change) of the PAGE, SESSION and APP objects in ECA rules.

Figure 20 shows an example of ECA rule with the use of the SESSION and APP objects. In the example, the application will retrieve the user information from the user identification (line 2) and the data and time from the application parameters (line 8).

```
    Example:
1           <!--%ECA load [main :: OnLoad]
2           SESSION->GetContentData(key = userId) AS result
3           OnSuccess
4           { ... }
5           OnFailure
6           { ... };
7
8           APP->GetContentData(key = sys_datetime) AS datetime
9           OnSuccess
10          { ... }
11          OnFailure
12          { ... };
13          %-->
```

Figure 20. Example of the SESSION and APP XANUI elements used in an ECA rule.

3.5.8 Validation

Designers can specify how to validate the values of the input widgets (only simple widgets, not containers) used in the ECA rules. The validation processes can be run while the application users input data in a widget (by means of dynamic constraints) or before the instantiation of an ECA rule (by means of static constraints). Figure 21 briefly describes the grammar of the validation constraints and shows a set of examples.

Validation constraints can be defined according to the expected data type of the value (e.g., for strings, the designers will use regular expressions). Moreover, in order to simplify the UI design, XANUI includes some of the expressions most frequently used in constraints (e.g., less than LT, greater than GT, field NOT EMPTY).

Validation constraints can be generic, meaning that they are not attached to a particular widget. XANUI defines the concept of constraint group (CHECKGROUP), with which designers can apply one or several generic constraints on one or several widgets.

Constraint groups can be also applied when the users interact with the widgets or before the execution of an ECA rule. However, the option taken for the whole group is applied to any individual constraint. Constraints are evaluated in the same order they were included in the group and the assessment process stops once a constraint check fails. With the CHECKALL option, designers can force the application to assess all the constraints.

Designers can also indicate which error messages will be visualised when the validation of a constraint or a group fails. In case of constraint groups, the final error message will be an aggregation of the error messages for the failed checks.

```
XANUI Syntax:
        (constraint)    <!--%CHECK ([WIDGET_NAME])?
                                TYPE \( [INT|FLOAT|BOOL|STRING|DOUBLE] \)
                                (RULE REGEX|NOTEMPTY|GT [INT|FLOAT]|LT [INT|FLOAT]|TRUE|FALSE)
                                [DYNAMIC|STATIC]
                                ERRORMSG [ERROR_MSG]
                                AS [RULE_NAME]%-->
        (group)         <!--%CHECKGROUP [RULE_NAME]+
                                APPLYON [WIDGET_NAME]+
                                [DYNAMIC|STATIC]?
                                [CHECKALL]?
                                ERRORMSG [ERROR_MSG]
                                AS [GROUP_NAME]%-->

Example 1 (constraint):
        <!--%CHECK textbox1 TYPE (STRING) RULE "^[A-Z]" STATIC ERRORMSG "Wrong format" AS rule1%-->

Example 2 (group):
        <!--%CHECK TYPE (STRING) RULE "^[A-Z]" STATIC AS rule2%-->
        <!--%CHECK TYPE (STRING) NOTEMPTY STATIC AS rule3%-->
        <!--%CHECKGROUP rule2 rule3 APPLYON textbox2 textbox3 ERRORMSG "Wrong format" AS group1%-->
```

Figure 21. Definition of validation rules using XANUI.

4 Generating Rich User Interfaces from XANUI Models

Once all the OOH4RIA models are completed, including the UI design with XANUI, designers can generate their RIA applications by means of two transformations "Generate RIA server" and

"Generate RIA client". The OOH4RIA process generates a RIA application applying the MVC architecture pattern (model-view-controller, see [8]), as depicted in Figure 22.

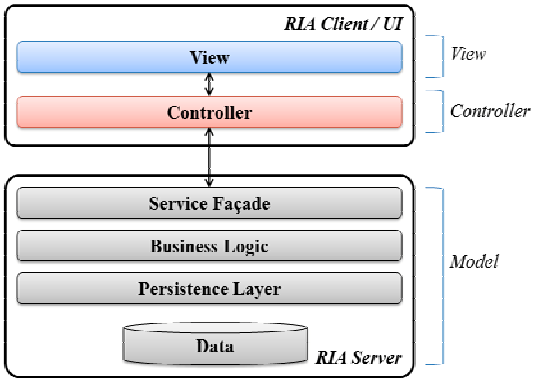

Figure 22. RIA architecture layers as generated by OOH4RIA.

In this pattern, the model (in grey) manages the application data, the view (in blue) is the interface to the user and the controller (in red) manages the view and its relationships with the model. In the basic RIA architecture generated by OOH4RIA, the RIA server is divided in four layers (implementing the model) and the client in two (implementing the view and the controller). Table 3 summarises the main functionalities of each layer.

Table 3. Description of the RIA architecture layers generated by OOH4RIA.

MVC Component	RIA Component	Architecture Layer	Description
Model	Server	Data	Database, stores the application data.
		Persistence	Manages the application data. Creates the main data objects.
		Business Logic	Performs the main functions of the application. Translates the internal data objects into data transfer objects for the Service Façade.
		Service Façade	Interface between the RIA server and client. Defines a set of Data Transfer Objects for information exchange.
Controller	Client	Controller	Manages the communication between the RIA server and client. Manages the behaviour of the user interface.
View		View	Manages the communication between the users and the application.

RIA servers and clients are generated in independent processes. Figure 23 shows the SPEM2 diagrams of the tasks "Generate RIA server" and "Generate RIA client".

The "Generate RIA server" transformation is a PIMToCode transformation (PIM – Platform Independent Model) with which designers can generate the code of the RIA server. From the Domain and Service models, the transformation generates the object-oriented business logic of the application. In order to support complex applications, the generated code applies the Domain Model pattern from EAPs (Enterprise Architectural Patterns [8]) for the business logic, and the Data Mapper pattern from EAPS for the persistence layer. From the Service model, the transformation generates an application façade component that filters the data and operations available in the server components to the client components.

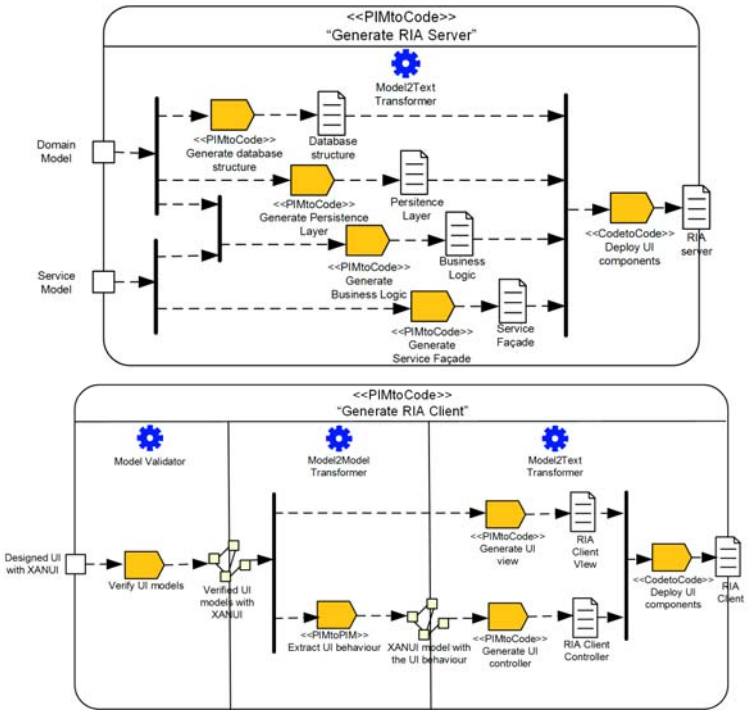

Figure 23. SPEM2 diagrams of the "Generate RIA Server" and "Generate RIA client" transformations.

From the XANUI UI models, the "Generate RIA client" transformation can generate the view and the controller of the RIA client. This transformation is performed by means of four main tasks:

1. Verify UI models: The XANUI models are analysed, thus ensuring that the UI components were annotated using the appropriate XANUI tags (e.g., the HTML <button> with the XANUI Button widget). This verification process is tailored to each UI technology (since the structural elements annotated are different from one technology to another).
2. Generate UI view: The XANUI model is analysed and all the XANUI tags are removed. XANUI widget/container annotations are replaced by global widget IDs in the original code, which will be used by the controller layer. The subcontexts are replaced by new UI regions, in which data will be loaded.
3. Extract UI behaviour: This task creates an intermediate XANUI model from the original set of UI models only containing the behaviour of the UI: ECA rules, validation constraints, and automatic context loading. It also merges name spaces and resolves the possible conflicts.
4. Generate UI controller: This task generates the UI controller from the elements of the intermediate XANUI model.

At present, an extension of OIDE (prototype) supports the XANUI generation process with the technologies and languages described in Table 4. The server transformations are implemented using the Xpand framework while the client ones use the Xtend2 framework because of its flexibility. In

order to include the annotations in the original UI code an editor was developed using the Xtext framework.

Table 4. Frameworks supported or under development in OIDE.

Component	Layer	Frameworks Supported	Languages Supported
RIA Server	Persistence	.NET (NHibernate), J2EE (Hibernate)	C#, Java
	Business Logic	.NET, J2EE	C#, Java
	Service Façade	.NET (WCF, WebAPI), Java (Axis2)	C#, Java
RIA Client	Controller	.NET (WPF), AngularJs	C#, JavaScript
	View	.NET (WPF), AngularJs	XAML, XHTML, CSS

Extending the support of XANUI towards new UI technology/framework requires the definition of new processes of code analysis and generation in the "Generation RIA client" activity, adapted to the target UI technology. In order to define these processes, the widgets of the new UI technology should firstly be mapped into XANUI structural elements. Those events and actions specific to the UI technology could be also defined as XANUI events and actions (in addition, new custom events and actions could be specified). With the same effort, the processes could be also implemented in a custom editor, which could assist the designers in the process of annotation, thus avoiding errors during the generation activities.

5 Using XANUI and OOH4RIA to Develop Rich Internet Applications

The XANUI DSL and the OOH4RIA development processes with XANUI were validated using real development projects. This paper will use the inaCatalog use case in order to demonstrate the feasibility of the proposals. The inaCatalog[a] system is a generic e-Commerce application that enables manufacturers 1) to share information about their products (e.g., shoes, ceramic figures, toys or fruits, among others) and 2) to sell any them directly to the final customers.

The section shows the manner in which both OOH4RIA processes, introduced in Section 2, can be applied in order to develop two different types of UI for the inaCatalog application:

1. A backend HTML view of the data available, with the bottom-up OOH4RIA process. This interface is needed by the administrators to manage the data entities (such as, families, orders, items, etc.) in the system.
2. An interface for the customers using the design-first OOH4RIA process. In this case, the UI will be designed for a WindowsRT tablet PC.

Both user interfaces will invoke the operations from a REST service-oriented façade based on the WebAPI framework.

[a] InaCátalog: http://www.inacatalog.com

5.1 Applying the Data-Intensive OOH4RIA process with XANUI

In the data-intensive process, the first step is the definition of the OOH4RIA domain model, which is a platform-independent model (inherited from the OO-H methodology) that represents the data entities with their properties (attributes and operations) and the relationships among them.

Figure 24 illustrates the Domain model for the inaCatalog application, which shows the entities of a typical e-commerce store. The application manages a set of products (Item class) that customers (*Customer* class) can buy through purchase orders (*Order* class). Customer can add as many order lines (*OrderDetails* class) as needed to each purchase order, one per product. Products can be classified into several categories (*Family* class), e.g. shoes, toys, etc. Moreover, inaCatalog can also manage the suppliers (*Supplier* class) for each of the products and the payments (*Payment* class) made for every order.

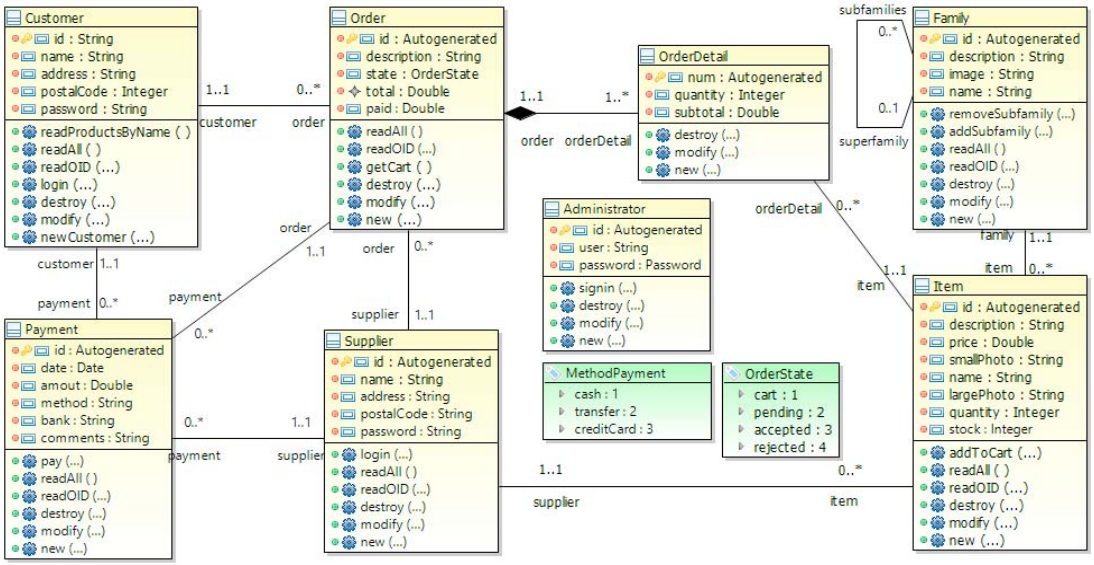

Figure 24. OOH4RIA Domain Model for the inaCatalog Rich Internet Application.

From the Domain Model, designers can generate the object-oriented business logic of the application by means of a model-to-text transformation. The generated code applies the domain model pattern of EAPS [8] and, more specifically, the code of the persistence layer applies the Data Mapper pattern using .NET NHibernate framework.

Once designers defined the Domain model, they can specify the data paths that will be used in the application in the OOH4RIA Service model. Figure 25 depicts the inaCatalogService model. According to the model, users will start visualising the information of the *AnonymousUser* service class. In order to access the data for the administrator, they need to log in by invoking the *signIn* operation (the invocation is represented by the service link with the same name).

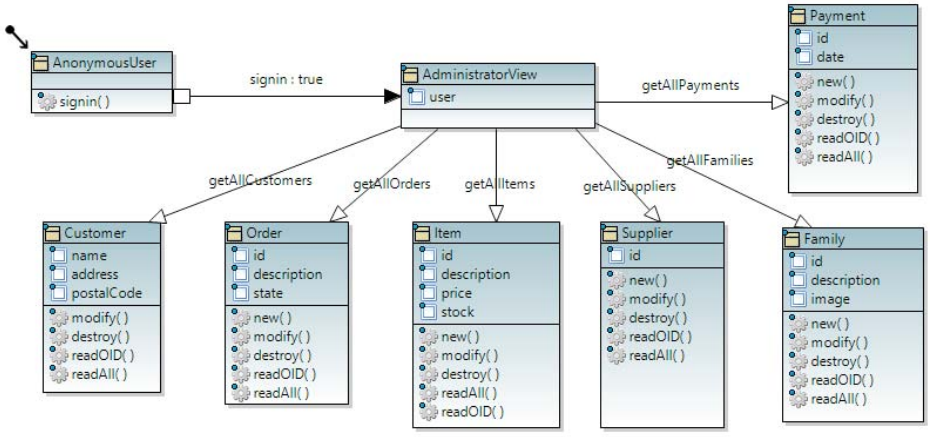

Figure 25. OOH4RIA Service Model for the administrator view of inaCatalog.

If the operation is successful, users can navigate to the private area of the application and thus retrieve the data shown in the *AdministratorView* service class. From this moment, users can act as the administrators to access to the backend functionalities. More specifically, administrators can access the data of any entity in the system (i.e., orders, items, customers, etc.) and they can create, modify and remove any type of entity.

At this point, using the Domain and the Service models, designers can invoke two transformations in the OOH4RIA process: 1) the "Generate RIA Server" transformation, and 2) the *Ser2Pres* transformation using the Service model as input.

The "Generate RIA server" transformation is a PIMToCode transformation with which designers can generate the code of the inaCatalog RIA server and from the Service model, the transformation generates an application façade component that filters the data and operations available in the server components to the client components. In this example, the AngularJS application requires a Service-Oriented Front-End Architecture (SOFEA) with a resource-oriented service interface in the server façade. In order to define the service interface, the transformation considers each service class as a resource (or resource type) and assigns a URI to it. Subsequently, it generates the HTTP operations (POST, GET, PUT and DELETE) from the service operations of each class. In addition, the data transfer objects sent between the server and the client sides are JSON objects, created from the service attributes and the traversal links.

The *Ser2Pres* transformation is a relevant contribution of this paper. It is a PIMToPIM transformation that, from the Service model, generates a default data-intensive user interface combining static code from a specific technology and XANUI annotations, which connect the UI with the server side. In this example, the static code generated is a set of HTML5 pages including CSS3 styles and using the Bootstrap responsive JavaScript framework. In this way, the designer can use any type of HTML editor to refine the aesthetics features on this default UI, e.g., the font style, colours, widget dimensions and spatial composition of different visual components. The XANUI annotations do not interfere in this process since they are embedded in special HTML comments.

Figure 26 shows an example of a XANUI model created from a HTML5 page, designed to add new items into the inaCatalog system. The main context of the Web page is the *AdministratorView*

service class (line 2), which contains the information about the administrator, and from which the administrator has access to all the information of the application through the traversal links. The page heading shows some user information such as the user name on the right side and, on the left side, there is a menu bar with the actions that the administrator can invoke over some data objects: Customer, Order, Item, Family, Offer and Payment. In addition, the screenshot shows a form to create new items. The form uses different input widgets: input text, input number, and text area. The designer defined the aesthetic features directly on the CSS style document.

As an example of use of subcontexts, in Figure 26, with the item subcontext (line 14), the designer specified that the application will retrieve all the items stored in the system using the *getAllItem* service (from the *Item* data object). Each HTML `<input>` element is associated to a XANUI Textbox element (lines 26, 35, 42), which establishes the data binding relationship with the elements of the Item service class. Moreover, the `<button>` HTML element is linked to a XANUI button element (line 57), which invokes the *newItem* ECA rule, managing the communication with the server. The result of the *newItem* ECA rule will be visualised in the XANUI Label element called information (line 62).

```
1  <!--%IMPORTS Shop.*%-->
2  <!--%CONTEXT AdministratorView @User%-->
3  <html lang="es">
4  <head>
5          <link rel="stylesheet" type="text/css" href="vendor/bootstrap/css/bootstrap.css"/>
6          <link rel="stylesheet" type="text/css" href="css/style.css"/>
7  </head>
8  <body class="container clearfix">
9          <div></div>
10         <!--%INCLUDE headingPanel FOR CONTEXT%-->
11         <div class="panel panel-default">
12                 <div class="panel-body">
13                         <form name="itemNew" class="form-horizontal" role="form">
14                 <!--%SUBCONTEXT item >> getAllItem%-->
15                         <!-- Inputs -->
16                         <div class="form-group">
17                                 <label class="col-sm-2 control-label">Id</label>
18                                 <div class="col-sm-10">
19                                 <p class="form-control-static"><em> </em></p>
20                                 <!--%LABEL id @Item.id%-->
21                                 </div>
22                         </div>
23                         <div class="form-group">
24                                 <label class="col-sm-2 control-label">Name</label>
25                                 <div class="col-sm-10">
26                                 <input type="text" class="form-control" placeholder="Name"/>
27                                         <!--%TEXTBOX name @Item.name-->
28                                 </div>
29                         </div>
30                         <div class="form-group">
31                                 <label class="col-sm-2 control-label">Description</label>
32                                 <div class="col-sm-10">
33                                         <textarea type="text" class="form-control"
34                                         placeholder="Description"/>
35                                         </textarea>
36                                         <!--%TEXTBOX description @Item.description%-->
37                                 </div>
38                         </div>
39                         <div class="form-group">
40                                 <label class="col-sm-2 control-label">Price</label>
41                                 <div class="col-sm-10">
42                                 <input type="number" class="form-control" placeholder="Price"/>
43                                         <!--%TEXTBOX price @Item.price%-->
44                                 </div>
45                         </div>
46                         <div class="form-group">
47                                 <label class="col-sm-2 control-label">Stock</label>
48                                 <div class="col-sm-10">
49                                 <input type="number" class="form-control" placeholder="Stock"/>
50                                         <!--%TEXTBOX stock @Item.stock%-->
51                                 </div>
52                         </div>
53                         <!-- Submit button -->
```

```
54                          <div class="form-group">
55                          <div class="col-sm-offset-2 col-sm-10">
56                          <button type="submit" class="btn btn-default">
57                                  New
58                          </button><!--%BUTTON newItem%-->
59                          </div><!--%ENDSUBCONTEXT%-->
60                          </form>
61                  </div>
62                  <div class="well">
63                          <!--%LABEL information%-->
64                  </div>
65                  </div>
66          </div>
67  </body>
68  </html>
```

Figure 26. HTML5 template of the New Item page

Once the design of the XANUI UI models is refined, the UI designer can invoke the last model-to-text transformation called “Generate RIA Client”, which generates the final user interface of the designed RIA. In this example, the transformation generates the code that connects the HTML5 templates to a resource-oriented RIA server side using the AngularJS JavaScript library. Figure 27 shows a functional *NewItem* Web page, which implements all the static and dynamic aspects designed with XANUI. The page heading shows 1) a button bar with which the administrations can invoke any CRUD operation for any entity and 2) a button with the name of the administrator and the actions to manage their preferences. Moreover, as mentioned before, the page content shows the *NewItem* form.

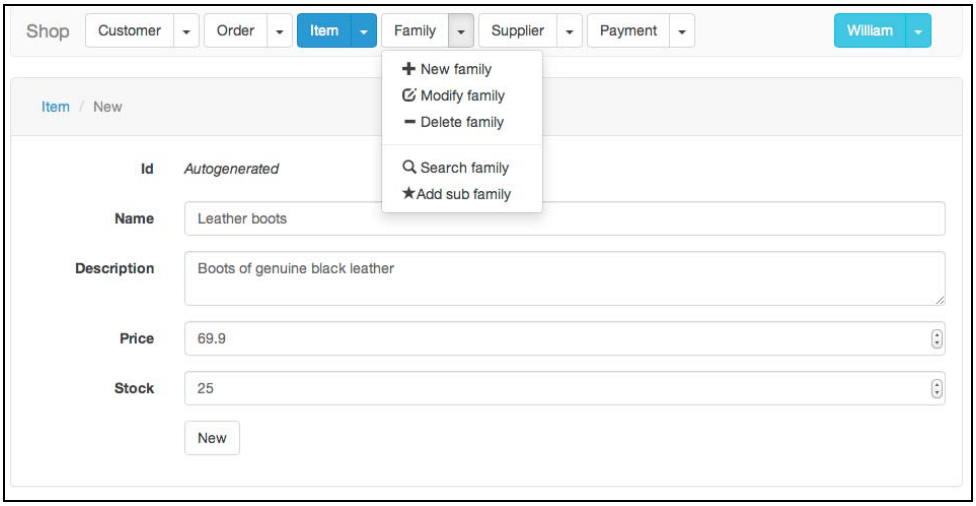

Figure 27. Screenshot of the NewItem page generated using AngularJS and Bootstrap.

5.2 Applying the Design-First OOH4RIA process with XANUI

The second OOH4RIA design process seeks the satisfaction of those stakeholders who are especially interested in the final UI design (UI structure and aesthetic features). This process also facilitates that designers capture the requirements of the stakeholders (related to the functionalities of the application and the look-and-feel and usability of the UI) and reduces the gap between the final application and the stakeholders’ expectations.

In this use case, the process starts from an existing rich UI, used directly by the final users, such as the one used by the suppliers in the catalogue application. Suppliers require a usable and user-friendly application to show their catalogues to their final customers and to track the status of different orders and payments. Usually, in order to obtain an attractive and user-friendly interface, designers use professional design tools such as Expression Blend, Dreamweaver, etc., to layout the different pages of the RIA.

As explained in Section 2, the OOH4RIA design-first process has two parallel activities. In the first one, the UI designer iteratively creates a set of different static UI templates gathering the information of the user requirements using a specific UI technology/framework. In the second activity, the modeller defines the Domain and the Service models in order to represent the business logic and the persistence layers of the Web application, as well as the service façade.

This example shares the Domain model from the previous section (see Figure 9). However, the Service model is different because the supplier view requires a different application façade that gives access to the catalogues and order operations.

Figure 28 shows the Service model for the supplier view. According to this model, users can visualise the application data anonymously through the *AnonymousUser* class. Subsequently, they should log in the application in order to access to the Supplier context. From this context, suppliers can obtain their catalogues with their items and product families. In addition, the supplier can manage the orders from different clients, create new orders and see their details. Each order has one or more payments, done by the customers using different payment methods.

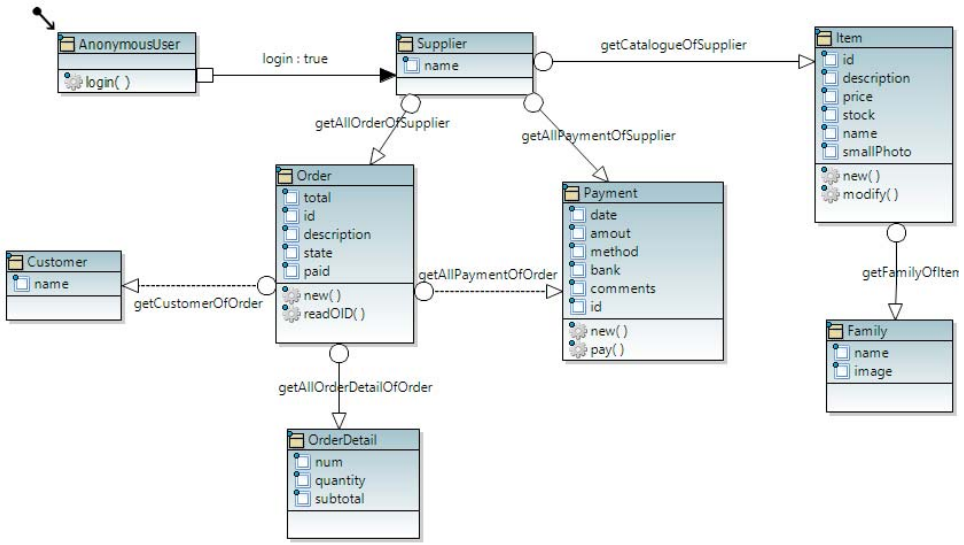

Figure 28. Fragment of the Service model for the suppliers.

Once both activities are finished and the Service model and UI templates are defined, the modeller can insert the XANUI model annotations into the designed templates in order to connect the UI templates to the server components. Due to the lack of space in the manuscript, Figure 29 only shows an excerpt of the XAML code of the inaCatalog for a Tablet PC, in which the style elements have been omitted. The figure shows the code that depicts the details about a specific order with a set of

payments that will be refunded by the customer. The XAML document starts with the definition of the Page element (line 4), which corresponds to a tablet page that realises a spatial distribution of the Widget elements rendered in a given moment. This page includes references to all the XAML libraries importing the style of widgets and external controls, such as the CalendarControl element (line 12).

```
1   <!--%IMPORTS inaCatalog.*%-->
2
3   <!--%CONTEXT Order @Order%-->
4   <Page x:Class="inaCatalogSupplier.Views.OrderAndPayments"
5           xmlns:local="using:inaCatalogSupplier.Views"
6           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
7           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
8           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10          mc:Ignorable="d"
11          xmlns:Controls="using:WinRTXamlToolkit.Controls"
12          xmlns:CalendarControls="using:System.Windows.Controls">
13
14          <!-- Order information, left side -->
15          <StackPanel Grid.Column="0" x:Name="DatosLiquidacionGridIzquierda">
16              <TextBlock Text="Total Order" Style="{StaticResource TituloCamposLiquidacionTextBlock}" />
17              <Grid Style="{StaticResource CampoLiquidacionGrid}">
18                  <TextBlock HorizontalAlignment="Right" Style="{StaticResource FieldTotalTextBlock}" />
19                          <!--%LABEL OrderTotal @Order.Total%-->
20              </Grid>
21
22              <TextBlock Text="Amount Paid" Style="{StaticResource TituloCamposLiquidacionTextBlock}" />
23              <Grid Style="{StaticResource CampoLiquidacionGrid}">
24                   <TextBox x:Name="paidStyle"
25                         LostFocus="txtCobrado_LostFocus"
26                         InputScope="CurrencyAmountAndSymbol"
27                         HorizontalAlignment="Right" VerticalAlignment="Center"
28                         FontSize="{StaticResource fontSize_L}"
29                         Style="{StaticResource TextBoxStyleSugerenciaGris}" />
30                           <!--%LABEL OrderPaid @Order.Paid%-->
31              </Grid>
32
33              <TextBlock Text="Customer" Style="{StaticResource TituloCamposLiquidacionTextBlock}" />
34              <Grid Style="{StaticResource CampoLiquidacionGrid}" Margin="0">
35                   <!--%SUBCONTEXT customerInfo >> getCustomerOfOrder2%-->
36                    <Grid.ColumnDefinitions>
37                          <ColumnDefinition Width="1*"/>
38                          <ColumnDefinition Width="Auto"/>
39                    </Grid.ColumnDefinitions>
40                    <TextBlock Text="Jose García"
41                           HorizontalAlignment="Right"
42                           Style="{StaticResource CampoFormaPagoTextBlock}" />
43                      <!--%LABEL CustomerName @Customer.Name%-->
44                    <Image Grid.Column="1"
45                           HorizontalAlignment="Center" VerticalAlignment="Center"
46                           Height="35" Width="35" Margin="0,0,10,0"
47                           Source="ms-appx:///Assets/eye_orange.png"/>
48                   <!--%ENDSUBCONTEXT%-->
49              </Grid>
50          </StackPanel>
51
52          <Grid Grid.Column="1" x:Name="OrderGridRight" Margin="30,0,0,0" . . .>
53              <Grid x:Name="PaymentListGrid" Grid.Row="1" Background="White" Margin="0,20" . . .>
54                      <!-- Heading -->
55                      <Grid x:Name="ListaCobrosCabecera" Grid.Row="0" Margin="10,0">…
56                         <TextBlock Grid.Column="0" Text="Id Payment"/>
57                         <TextBlock Grid.Column="1" Text="Due Date" HorizontalAlignment="Center" />
58                         <TextBlock Grid.Column="2" Text="Paid" HorizontalAlignment="Right" />
59                         <TextBlock Grid.Column="3" Text="Doubt" HorizontalAlignment="Right"/>
60                         <TextBlock Grid.Column="4" Text="Payment Date" HorizontalAlignment="Center" />
61                      </Grid>
62
63                      <Grid Grid.Row="1">
64                       <!--%SUBCONTEXT paymentInfo >> getAllPaymentOfOrder2%-->
65                          <TextBlock Grid.Column="0" Text="1 - 3434"/>
66                          <!--%LABEL PaymentId @Payment.Id%-->
67                          <TextBlock Grid.Column="1" Text="12/12/2014" HorizontalAlignment="Center"/>
68                          <!--%LABEL PaymentDueDate @Payment.DueDate%-->
69                          <TextBlock Grid.Column="2" Text="350" HorizontalAlignment="Right"/>
70                          <!--%LABEL PaymentPaid @Payment.Paid%-->
71                          <TextBlock Grid.Column="3" Text="150" HorizontalAlignment="Right"/>
72                          <!--%LABEL PaymentDoubt @Payment.Doubt%-->
73                          <TextBlock Grid.Column="4" Text="22/11/2014" HorizontalAlignment="Center"/>
```

```
74                          <!--%LABEL PaymentDate @Payment.Date%-->
75                      <!--%ENDSUBCONTEXT%-->
76                    </Grid>
77              </Grid>
78  </Page>
```

Figure 29. WindowsRT XAML view of Order Information Page

The main context, attached to the Page XAML element and linked to the *Order* service class (line 3), represents the information about an order, the customer and the payment information (also using the service associations). The page contains several TextBlock elements (similar to labels) used to show the information about the order (e.g., line 18 and 22). The designer added a XANUI Label element after each TextBlock XAML element in order to bind the value of the Order properties (such as total, paid, customer name, etc.) to the XAML widgets (see lines 19, 30 or 43). Furthermore, the page contains a list of the payments for the order (lines 63-76), obtained from the *getAllPaymentsOfOrder2* traversal link.

Figure 30 depicts the final user interface of the payment view of the supplier for a Tablet PC.

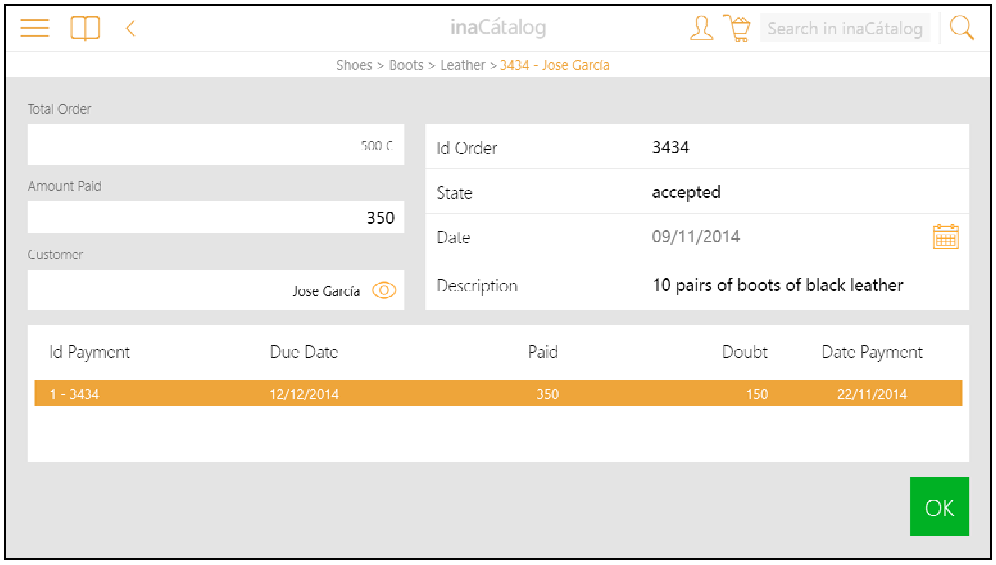

Figure 30. Screenshot of the interface of order management.

6 Related Work

As described in the introduction, current model-driven methodologies share some limitations when they address the design and development of the user interface. This section compares the solutions adopted by the main model-driven development methodologies with the XANUI solution applied to OOH4RIA. More specifically, the following solutions will be analysed in the paper: WebML, RUX-Method, MockupDD and IFML.

6.1 Summary

Table 5 shows a summary of the results of the analysis. The analysis is focused on the features related to their models for designing the UI and the development processes they propose.

Table 5. Summary of the analysis of the model-driven solutions for UI development.

Feature	**Solution**				
	WebML	**RUX**	**MOCKUPDD**	**IFML**	**OOH4RIA**
Development Process	Data-intensive	Data-intensive	Prototype-Driven	Can be integrated in a data-intensive process	**Data-Intensive / Design-first**
UI models		Abstract/ Concrete Presentation Model	UI mock-up	-	**XANUI**
Concrete syntax of the UI Model	Graphical	Graphical	Graphical	Graphical	**Textual**
UI Model represents UI structure	Yes	Yes	Yes	Yes	**No, the structural annotations only enable the definition of the data binding properties and the UI behaviour.**
UI Model represents the UI aesthetic features	No	No/Yes	No	No	**No, these properties remain in the UI code.**
UI model represents UI behaviour	Yes	Yes	Yes	Yes	**Yes**
UI model defines event-driven UI behaviour	Yes	Yes		Yes	**Yes**
Is UI model platform-independent?	Yes	Yes/No	Yes	Yes	**Yes**
UI model can be visualised with design tools	No	No	No	No	**Yes**
UI model defines UI templates.	No	No	Yes	Yes	**Yes**
UI model supports data binding.	Yes	Yes	No	Yes	**Yes**

6.2 WebML

During the last decade, the original WebML methodology has been extended with primitives for modelling different aspects of Web applications or even new application types. Bozzon et al. [4] and

Fraternalli et al. [9] introduced an extension of the method to tackle the development of Rich Internet Applications. Specifically, they proposed an extension of the Data model to facilitate the representation of distributed and temporary data objects, and the Hypertext model, which include new primitives to manage the new types of data objects. Moreover, the authors propose to include a new model for defining the dynamic behaviour of the elements of the RIA user interface, i.e., the RIA dynamic model, similar to UML activity diagrams.

The WebML proposal for rich user interfaces represents the structural and behavioural aspects of the UI. The authors proposed a graphical platform-independent model capable of capturing these features, but, it is not focused on the aesthetic aspects of the design. The WebML presentation model supports the definition of data binding and event-driven rules for managing the interaction with the users. It does not support the creation and instantiation of templates.

While the WebML methodology only proposes a data-intensive development process, with XANUI, OOH4RIA can offer two processes to define the user interface: a design-first process, focused on the design of the UI from the first design phases; and the original data-intensive process. In addition, the WebML presentation model represents the UI conceptually and does not offer view that could be validated by a non-expert customer. In order to create a prototype of the UI, the WebML designers should generate the code first from their conceptual UI model and, subsequently, the UI designers should adapt the generated code.

6.3 RUX-Method

RUX-Method – Rich User eXperience Method [12, 16] – is a model-driven methodology focused on the development of rich user interfaces, i.e., the user interface and the client modules of a Rich Internet Application. This methodology proposes a three-step process driven by the specification of three models:

- the Abstract Interface, which represents the abstract elements of the interface, independent from the chosen implementation technology;
- the Concrete Interface, which represents the structure and the behaviour of the RIA interface in a platform-independent manner. It is based on three presentations: Spatial (structure of the interface and aesthetic features), Temporal (behaviour of the interface) and Interaction (user interactions). It relates the elements of the interface to the services provided by the RIA server.
- the Final Interface, which adapts the abstract elements of the two previous interfaces to a specific technology and relates the elements of the interface with the services provided by the RIA server.

RUX-Method does not support the design and development of the RIA server modules since it is focused on the RIA user interface and client modules. The RUX process and models are meant to be integrated into the WebML data-intensive process.

RUX-Method offers three different graphical models to create a new UI: one platform-independent, the Abstract Interface, and two platform-specific, the Concrete and the Final Interface. They represent the main structural and behavioural aspects of the user interface (spatial arrangement,

event-driven rules or data binding). The aesthetic properties can be also captured by this set of diagrams. The RUX models cannot define UI templates.

Designers need the RUX-tool (which implements the RUX-Method methodology) in order to obtain a view of the final user interface using these models. With the XANUI models in OOH4RIA, designers can use any of the existing industrial tools to represent the structure and the aesthetic features of the UI. In addition, XANUI models could be combined with existing design templates (for instance, available online or from another project) from the first stages of development. Both aspects combined should result in UI templates closer to the final UI. The resulting templates can be a relevant source of feedback from the stakeholders.

Another difference between OOH4RIA and RUX-Method is that the later reuses the server models of the WebML approach, which follows a data-intensive approach.

6.4 MockupDD

MockupDD [17] is an approach that defines a prototype-driven process over the UWE models. This method is focused on gathering the user requirements through the specification of UI mock-ups. The UI mock-ups can be translated into the UWE presentation model and subsequently, transformed to the UWE Navigational and Domain models through a set of heuristic rules.

Mock-ups are templates that can represent the different views of the UI and their components, i.e., the UI structure, as well as their behaviour when the user interacts with them. However, they cannot capture the definition of aesthetic features directly. In order to capture these properties, the mock-ups are transformed into the UWE presentation model, which can represent them.

MockupDD is an example of how a prototype-driven process could be implemented in the current model-driven methodologies. However, mock-ups are incomplete and could be helpful in the early stages of development but, when the design is in its later stages, designers should use the presentation model generated from them. In this case, designers would need two specialised tools: one for the design of the mock-ups and another for the refinement of the presentation models.

In MockupDD, mock-ups can be defined using tools such as Balsamic or MockFlow, which are oriented to the communication with the customers. For this reason, some UI features such as widget composition cannot be captured completely. For instance, the transformation from the mock-ups to the UWE results in an abstract UWE Presentation model, not close to the final UI. In addition, the UWE model cannot be combined with UI design templates.

Unlike MockupDD, XANUI does not follow a complete prototype-driven development process since the Domain model is not created from the UI but in a separate process. The design process of the UI can have a negative influence in the design of the data objects needed in the application server (e.g., redundancy of information, suboptimal separation of concerns, ambiguous semantics, etc.) Moreover, domain models can represent more information in the data objects which are not always visualised in the UI (for instance, the meta-information of the data objects: internal IDs, life time, etc.; or the internal information of the application). For these reasons, with OOH4RIA, the Domain model and UI can be specified in parallel. The Service model would be the only model that could be automatically inferred from the previous ones.

6.5 IFML

IFML (Interaction Flow Modelling Language, [5]) is a recent OMG standard for the development of user interfaces proposed by the same authors as WebML. With this new language, designers can represent the UI structure and the interaction between the UI and its users, and between UI components. IFML provides design primitives to represent the spatial disposition of the UI components and their behaviour based on the user events. However, IFML does not represent the aesthetic aspects (styles, colours, etc.) of the UI, which will be arranged by the UI designer once the code is generated. The resulting design is independent from the final technology and platform for which the UI will be generated.

IFML can be integrated only with data-intensive processes and domain models, e.g., ER, UML class diagrams, etc. At present, it can be used along the WebML processes and models implemented in the WebRatio tool.

Unlike XANUI, IFML can only be integrated in data-intensive development processes. Moreover, IFML provides a PIM presentation model (similar to the WebML one) that hides aesthetic features and cannot be combined with real design templates. Therefore, non-expert customers could not validate the UI resulting from the model directly. The modeller should generate the UI, and subsequently the UI designers should adapt the generated code.

7 Discussion

In Model-Driven Web Engineering, UI technologies evolve faster than the model-driven methodologies, the cost of creating/maintaining code generators for every UI framework is high and in order to facilitate the adoption of the solutions by designers, mature model editors are needed even though the cost of development is also high. Furthermore, most of the methodologies for RIA development follow a data-intensive process. However, in some cases, in order to capture adequately the requirements of the UI from the stakeholders, UI prototypes or mock-ups are useful from the first stages of design.

The MDE solutions normally propose graphical UI models that aim at minimising the gap between the final UI and the models represented. These models capture the information regarding the UI structure (i.e., the widget spatial disposition) and the interaction with the users (i.e., the behaviour). However, the aesthetic features can be changed either a) after the generation of the code, with the risks associated to code regeneration; or b) with the editors provided by each solution, which normally offer less features than a professional editor, adapted to one specific technology. Therefore, in the second option, designers need to learn how to work with a new editor and subsequently complete the rich UI after the generation of the UI code with their preferred editor.

This is a problem shared by the OOH4RIA Presentation and Orchestration models, which is a Platform-Specific model (PSM) for the design of RIA clients. XANUI was developed based on the experience obtained in the development of these model. The goal of the XANUI solution is to provide consumers a high quality view of the final UI from the early stages of design. With XANUI, designers can define their UI templates using a specific UI framework and their professional tools and, subsequently, link them to the RIA server components, which could be done by another

designer/engineer. Unlike other similar model-driven approaches, XANUI can provide to the stakeholders UI models with higher quality and closer to the final UI.

From the MDE perspective, XANUI is a platform independent model, since their primitives are technology independent, with a textual syntax. Due to its syntax, the XANUI primitives can be embedded/combined in software code that is heavily dependent from the technology. In this way, the level of abstraction from the code is lower but it should be also easier to design the UI and to interact with the stakeholders.

The distinctive feature of XANUI is its capability of inserting in any XML-based UI language. This feature introduces the need of understanding the code in which XANUI is embedded and the relationships between the XANUI annotations and the elements of the language. Therefore, apart from the code generator, a reader is needed (i.e., a read-only editor). Another possible strategy could be to consider XANUI annotations as protected regions. In this way, the code generator would only need to replace the annotations by their corresponding code. However, this could also make XANUI models not compatible with other design tools for UI design, and thus lose one of the main benefits.

OOH4RIA can use XANUI in two development processes: a data-intensive or a design-first process, which can be adapted to the needs of the application or the stakeholders. In each of them, XANUI is used in a different manner. While in the bottom-up process, the XANUI annotations can be partially generated using a model transformation, in the design-first process, all the annotations should be manually embedded in the code. The choice of the design-first process should take into account, apart from the benefits in UI design and requirement elicitation, the fact that the required effort of development in the second case can be higher. This effort could be minimised by improving the usability of the XANUI editor, which would imply other type of costs.

XANUI is a new textual model for UI design that could be applied to several model-driven methodologies. In order to represent the UI, XANUI reuses and extends some concepts and primitives already existing in the OOH4RIA UI models (i.e., Presentation and Orchestration models). However, XANUI does not deal with the representation of service interfaces nor data objects and designers therefore need to specify this information by means of other models, which, in this paper, are the OOH4RIA Domain and Service models. The design-first OOH4RIA process highlights the benefits of XANUI and, at the same time, overcomes the limitations of the original OOH4RIA process regarding the UI design.

Regarding the syntax, XANUI is not a closed language and can already evolve in some aspects. For instance, custom widgets could be defined in order to adapt the language to new (or not frequently used) frameworks. The extensibility of the language could be improved with custom widgets. In validation constraints, error messages can be better managed and associated to a specific widget. These are issues that will be addressed in a new version of the language.

At present, the only tool supporting XANUI is the OIDE tool, which provides an Xtext editor for creating XANUI models and the code generators for some UI frameworks. However, the usability of the editor can be also improved. At present, it does not interpret the software code on which the XANUI code is embedded and thus, for instance, it does not provide syntax highlighting. Regarding the code generators, OIDE generates code for AngularJS from XANUI code and the .NET generator (XAML and C#) is a prototype being tested.

8 Conclusions and Future Work

This paper introduced the XANUI domain specific language for the development of rich UIs and the application of this language to two separate design processes of the OOH4RIA methodology.

Using the XANUI DSL, UI designers will be able to combine the UI designs from professional tools with the model-driven methodologies for the development of the application façade. The combination of OOH4RIA and XANUI offers a flexible development process for the design of RIAs and mobile applications. The new design-first OOH4RIA development process with XANUI is effective for the early elicitation of the UI requirements and allows designers to work independently from the design of the server components.

The XANUI language captures most of the mechanisms already used in the definition of the UI structure: simple and complex widgets, data binding, data contexts and scopes; and the UI behaviour: ECA rules for the management of the user interactions based on the events they raise. The solution is supported by the OIDE tool, which provides the XANUI editor and the code generators for HTML and XAML.

The paper showed an example of the application of OOH4RIA with XANUI to a real project for the development of a catalogue application. However, in order to further demonstrate the features of XANUI, empirical experiments could be performed with different types of designers, from students to senior designers/developers.

The following lines of future work can be defined from this paper: a) to study the integration of XANUI with other model-driven methodologies for the development of RIAs; b) to improve the usability of the XANUI editor in order to minimise the possible errors of the designers; c) to empirically assess the different features of the model and the quality of the language; and d) to develop code generator for other JavaScript-based user interface technologies such as React or Backbone.js and other XML-based UI mobile frameworks such as Android and iOS.

References

1. Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M. and Shuster, J. E. UIML: an appliance-independent XML user interface language. Computer Networks, 31(11), 1695-1708 (1999).
2. Berti, S., Correani, F., Mori, G., Paternó, F. and Santoro, C., TERESA: A Transformation- Based Environment for Designing Multi-Device Interactive Applications. In proceedings of CHI 2004, CHI 2004 extended abstracts on Human factors in Computing Systems. (New York, 2004)
3. Boedcher, A., Mukasa, K. and Zuehlke, D., Capturing Common and Variable Design Aspects for Ubiquitous Computing with MB-UID. In proceedings of the MDDAUI 2005, held with MoDELS 2005 (Jamaica, 2005)
4. Bozzon, A., Comai, S., Fraternali, P. and Carughi, G.T., Conceptual modeling and code generation for rich internet applications. In proceedings of the ICWE 2006 (New York, 2006).
5. Brambilla, M., IFML: Building the Front-End of Web and Mobile Applications with OMG's Interaction Flow Modeling Language. In proceedings of the ICWE 2014 (Toulouse, 2014)

6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996.
7. Casteleyn, S., Garrigós, I. and Mazón, J. Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. ACM Transactions on the Web, 8(3), 1-18 (2014)
8. Fowler, M. Patterns of enterprise application architecture. Addison-Wesley, 2002.
9. Fraternali, P., Comai, S., Bozzon, A. and Toffeti Carughi, G. Engineering rich internet applications with a model-driven approach. ACM Transactions on the Web, 4. 1-47 (2010).
10. Gómez, J., Cachero, C. and Pastor, O. Conceptual Modeling of Device-Independent Web Applications. IEEE MultiMedia, 8. 2-39 (2001).
11. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. and López-Jaquero, V. USIXML: a language supporting multi-path development of user interfaces. In Bastide, R., Palanque, P. and Roth, J. eds. Engineering human computer interaction and interactive systems, Springer Berlin Heidelberg, 2005, 200-220.
12. Linaje Trigueros, M., Preciado, J.C. and Sánchez-Figueroa, F. Engineering Rich Internet Application User Interfaces over Legacy Web Models. IEEE Internet Computing 11, 53–59 (2007).
13. Meliá, S., Gómez, J., Pérez, S. and Díaz, O., 2008. A model-driven development for GWT-based rich Internet applications with OOH4RIA. In proceedings of the ICWE 2008 (New York, 2008).
14. Meliá, S., Gómez, J., Pérez, S. and Díaz, O., Architectural and technological variability in Rich Internet Applications. IEEE Internet Computing 14, 24–32 (2010).
15. Pérez, S., Díaz, O., Meliá, S. and Gómez, J. Facing interaction-rich RIAs: The orchestration model. In proceedings of the ICWE 2008 (New York, 2008).
16. Preciado, J.C., Linaje, M. and Sanchez-Figueroa, F. Enriching model-based web applications presentation. Journal of Web Engineering, 7(3). 239-256 (2008)
17. Rivero, J. M., Grigera, J., Rossi, G., Luna, E. R. and Koch, N., Improving Agility in Model-Driven Web Engineering. In Proceedings of the CAiSE Forum (2011).
18. Robles, E., Grigera, J. and Rossi, G., Bridging test and model-driven approaches in web engineering. In proceedings of the ICWE 2009 (San Sebastian, 2006).