Journal of Symbolic Computation 46 (2011) 207–217

Contents lists available at ScienceDirect

Journal of Symbolic Computation

journal homepage: www.elsevier.com/locate/jsc

# Providing RIA user interfaces with accessibility properties☆

Marino Linaje, Adolfo Lozano-Tello, Miguel A. Perez-Toledano, Juan Carlos Preciado, Roberto Rodriguez-Echeverria, Fernando Sanchez-Figueroa [1]

*Quercus SEG, Universidad de Extremadura, Av. Universidad s/n 10003, Spain*

ARTICLE INFO

*Article history:*
Received 15 June 2010
Accepted 21 July 2010
Available online 27 August 2010

*Keywords:*
RIA
Accessibility
Ontologies
User interfaces
Model-driven development

ABSTRACT

Rich Internet Applications (RIAs) technologies are challenging the way in which the Web is being developed. However, from the UI accessibility point of view, these technologies pose new challenges that the Web Accessibility Initiative of the W3C is trying to solve through the use of a standard specification for Accessible Rich Internet Applications (WAI-ARIA). Currently, the introduction of properties defined in WAI-ARIA is being done in an ad-hoc manner due to the lack of models, methodologies and tools to support the design of accessible RIA UIs. In this paper we propose a semantic approach to deal with this modeling issue by extending the RUX-Method, a model-based method to build RIA UIs. The approach includes the validation process of the accessibility issues at two different levels: the UI structure and the interactions behavior.



## 1. Introduction

During the last few years, developers are switching to novel technologies known under the collective name of Rich Internet Applications. RIAs combine the benefits of the Web distribution architecture with the UI interactivity and multimedia support of desktop environments.

From the UI point of view, RIAs expand traditional Web UI features, providing homogeneous multimedia contents; customized controls (e.g., accordion, modal window, etc.) to expand the

☆ This work has been partially supported by Spanish projects: TSI-020501-2008-47 (granted by Ministerio de Industria) and TIN2008-02985 (granted by Ministerio de Ciencia e Innovacion).

*E-mail addresses:* mlinaje@unex.es (M. Linaje), alozano@unex.es (A. Lozano-Tello), toledano@unex.es (M.A. Perez-Toledano), jcpreciado@unex.es (J.C. Preciado), rre@unex.es (R. Rodriguez-Echeverria), fernando@unex.es (F. Sanchez-Figueroa).

[1] Fax: +34 927 257202.

0747-7171/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2010.08.008

possibilities of the standard HTML controls (e.g., textinput, combobox, etc.); animations, transition effects and so on; advanced interaction support through Web-extended events in the widgets (e.g., drag-and-drop or custom events). Finally, they support the single-page paradigm, where the UI is composed by elements that can be individually loaded, displayed and refreshed according to the UI requirements. All these new capabilities introduce new accessibility challenges on the Web.

To provide an accessible user experience to people with special needs, assistive technologies (ATs) need to be able to interact with the UI. Some RIA technologies provide support for accessibility guidelines e.g., WCAG.[2] However, these guides were created for traditional Web UIs and they do not support RIA UI accessibility. WAI-ARIA[3] tries to solve this problem adding semantics about these new capabilities to HTML. However, to our knowledge the introduction of the WAI-ARIA accessibility features is currently being done in an ad-hoc manner due to the lack of models and methodologies to support the RIA UI design[4] (Borodin et al., 2008; Stringer et al., 2007; Chen and Raman, 2008; Lunn et al., 2009) .

Precisely, this lack of methodologies for RIA UI design was the origin of the RUX-Method (Linaje et al., 2007), a method for enriching the UI of model-based Web Applications with RIA features. However, in previous works we did not consider accessibility issues. Even when the WAI-ARIA is still a working draft, in this paper we show a RUX-Method extension to consider the inclusion of accessibility issues according to this draft due to the many advantages that it has already introduced for accessible RIA UIs. This extension is done applying the lessons learnt in the SAW (System for Accessibility to the Web) project (Sánchez-Figueroa et al., 2007). The bridge between the RUX-Method and SAW relies on the Component Library provided by the former and the use of ontologies provided by the latter.

The rest of the paper is as follows. In Section 2 the accessibility issues in RIAs are considered. Section 3 gives an overview of the RUX-Method. In Section 4 the introduction of accessibility in the RUX-Method is shown. Finally, Section 5 summarizes the paper and presents several considerations about related works.

## 2. Accessibility in Rich Internet Applications user interfaces

RIA UIs introduce two new important elements to traditional Web applications: (1) custom controls also called widgets (e.g., slider) to overcome the limited number of controls available in HTML, and (2) asynchronous communications to permit requests to the server allowing partial refreshments of the UI. These elements introduce quite a few particularities to make the UI understandable by an AT:

(1) ATs are not able to take the control of the widget to allow users to use the keyboard as the only input device, avoiding the mouse. In Web 1.0 applications, only the elements *A*, *AREA*, *BUTTON*, *INPUT*, *OBJECT*, *SELECT* and *TEXTAREA* (ie, links, controls and external objects) can receive the focus.

(2) When part of the UI is updated, the AT cannot capture this UI action and cannot notify the user about this change.

(3) ATs do not know anything about the widget purpose (what is it for).

(4) AJAX widgets usually define their own states and properties out of the HTML standard (i.e., using Javascript), so the ATs cannot use these properties and states due to ATs not knowing even the existence of a specific property or state.

(5) It is more difficult than in traditional Web applications to differentiate sections (e.g., navigation section) within the same page due to the lack of differences in their mark-up specification.

2 http://www.w3.org/TR/WCAG20/.

3 http://www.w3.org/TR/wai-aria/.

4 http://developer.yahoo.com/yui/examples/container/container-ariaplugin.html.

### 2.1. WAI-ARIA

The incorporation of WAI-ARIA to the W3C web standards provides a way to include proper type semantics on RIA UIs to make them accessible, usable and interoperable with ATs. Among others, WAI-ARIA identifies the types of widgets and structures that are recognized by accessibility products, by providing an ontology.[5]

To solve the problems listed before, WAI-ARIA introduces several concepts:

(1) To solve the keyboard navigation, WAI-ARIA introduces an extension to the use of the *tabindex* property already available in HTML, making this attribute able to be present in any visible element of the UI. When the *tabindex* property is set to −1, it implies that the element can only receive the focus using JavaScript, so for example in menu widgets, the menu item *tabindex* can be set to −1 and javascript can be used to define keyboard events to navigate the menu items without a pointer.
(2) To solve the notification of partial UI updates, WAI-ARIA introduces live regions (with several attributes like *aria-live*, *aria-busy*…) that can be associated to any widget in order to specify when the widget(s) is able to update itself totally, partially or never (according to the value of the specific properties). Using live regions any update on the screen can be caught independently of its condition e.g., a structure, style or data update, and its source e.g., server pushing, user interaction or temporal behavior.
(3) To know the purpose of an instance of a widget, WAI-ARIA creates the *role* attribute. It allows elements with a given role to be understood as a particular widget or structural type regardless of any semantic inherited from the implementing technology. Roles are a common property of accessibility APIs used by applications to support ATs. The role taxonomy currently includes interaction widget (UI widget) and structural document (content organization) types of objects.
(4) To support custom widget properties and states, WAI-ARIA defines changeable states and properties of elements. States and properties are used to declare important properties of an element that affect and describe interactions. These properties enable the user agent or operating system to properly handle a given element even when these properties are altered dynamically by scripts. For example, alternative input and output technology such as screen readers must recognize if an object is disabled, checked, focused, collapsed, hidden, etc. The available states for a widget depend on the *role* value assigned to the widget.
(5) To differentiate regions within the UI, WAI-ARIA uses the concept of Document Landmarks that, at the implementation level, are roles (in fact, they also use the role property). This role applied over a set of widgets serves to specify when the region contains a set of banners, navigational elements, etc. These landmarks allow localizing sections faster, skipping the irrelevant elements for the next user interaction.

## 3. RUX-Method overview

The RUX-Method (Linaje et al., 2007), is a model driven method which supports the design of multimedia, multi-modal and multi-device interactive Web 2.0 UIs for RIAs. The RUX-Method can be combined with many Web models (Rossi et al., 2008) which model the data and business logic in order to build complete rich Web Applications.

The RUX-Method, depicted in Fig. 1, is broken down into four design levels: Concepts and Tasks, Abstract UI, Concrete UI and Final UI. The RUX-Method takes Concepts and Tasks (i.e., data and business logic) from the underlying Web model.

In the RUX-Method, the Abstract UI provides a conceptual representation of the UI with all the features that are common to all the RIA devices and development platforms, without any kind of spatial, look&feel or behavior dependencies. Following this idea, each component of the Abstract UI is also independent from any specific RIA device and rendering technology. In the Concrete UI, the UI

[5] http://www.w3.org/WAI/ARIA/schemata/aria-1.rdf.

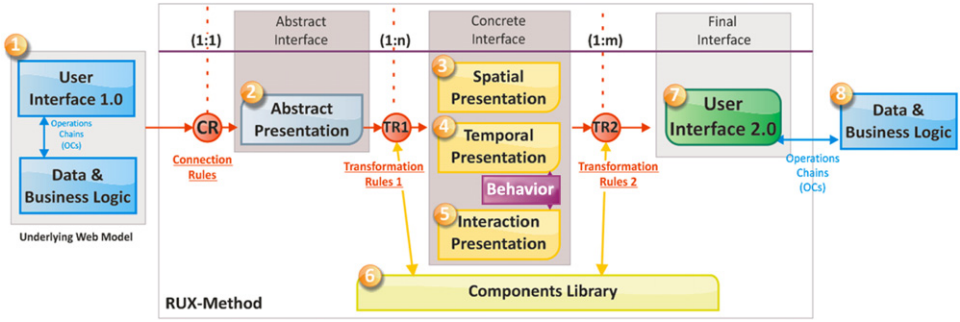


**Fig. 1.** RUX-Method architecture overview.

can be optimized for a specific device or a set of devices. Concrete UI is divided into three Presentation levels in order to provide a deeper separation of concerns according to the Web UIs requests: Spatial, Temporal and Interaction Presentation. Spatial Presentation allows the spatial arrangement of the UI to be specified, as well as the look&feel. Temporal Presentation allows the specification of those events which require a temporal synchronization (e.g. animations), while Interaction Presentation allows modeling those events that the user produces over the UI (e.g. clicking a button). Connected to both is the Behavior model which specifies the actions to carry out under certain conditions. Behaviors can be triggered by temporal or interaction events.

The RUX-Method process ends with the Final UI specification which provides the code generation of the modeled application when using the RUX-Tool. The code generated by the RUX-Tool is specific for a device or a set of devices and for an RIA development platform. This code is deployed together with the data and business logic code generated using other CASE Tools (e.g., WebRatio).

There are two kinds of adaptation phases in the RUX-Method according to the UI levels defined above. Firstly, the adaptation phase that catches and adapts Web data, contents and navigation from the underlying Web model to the RUX-Method Abstract UI are called Connection Rules. Secondly, the adaptation phase that fits this Abstract UI to one or more particular devices and grants a right access to the business logic are called Transformation Rules 1 (TR1). Finally, there is an additional transformation phase supporting and ensuring the right code generation (Transformation Rules 2 or TR2 in Fig. 1).

Closely related to the Transformation Rules, is the Component Library. Each RUX-Method UI Component specification is stored in this library. The library also stores how the transformations among components of different levels will be carried out.

The workflow of the behaviors (marked Behaviors in Fig. 1) between components at the concrete UI level and other tiers (e.g., business logic) is specified using the BPMN[6] standard. The main interest for this paper relies on both, the Component Library and the workflow specification for user and temporal interactions. Next, we briefly explain both.

### 3.1. Component library in the RUX-Method

The Component Library is responsible for: (1) storing the component specification (mainly composed of name, methods, properties and events), (2) specifying the transformation capabilities for each component from an UI level into other components in the following UI level and (3) keeping the hierarchy among components at each UI level independently from other levels. For a given component several transformations can be defined depending on the target components that can be *N*; e.g., an Abstract UI component is *Text* whose type can be *input* or *output*; this component could be transformed to the *RichTextEdit* or *TextControl* Concrete UI components or to any other component

6 http://www.bpmn.org.

that the designer decides to integrate in the Library. Components can be defined extending other components. The *BaseComponent* is the root of the hierarchy of components.

In the RUX-Method the Component Library also establishes the mapping among components of adjacent UI levels, focusing on its properties, methods and events. These mappings are indeed the base of transformation rules (TR1 and TR2) among components by means of skeletons as fully explained in Linaje et al. (2008). In Linaje et al. (2007) the set of RUX native components[7] was fixed and categorized into controls, layouts and navigators.

For clarification next we show an excerpt of the Spatial Presentation (at the Concrete UI level) where the Components are instantiated. This excerpt includes an instance of the *AutoCompleteTC* component which belongs to the Component Library. The component is identified by the unique key *CTMI1*, whose origin is a *Text Media Input* from the Abstract UI level, identified by the unique key *ATMI1*. The node Properties identifies a set of properties related with a specific component or type of components. The last property specifies the role that this component plays from the accessibility point of view (explained in Section 4) and takes a WAI-ARIA compliant value.

Example of an instantiated component at the Concrete UI level.

```
<spatialPresentation>
  <structure>
    <part id="CTMI1" class="AutoCompleteTC" source="ATMI1"/>
  </structure>
  <properties id="Style1" target="AutoCompleteTC">
    <property name="vAlign">center</property>
    <property name="hAlign">center</property>
    <property name="width">50%</property>
    <property name="height">50%</property>
    <property name="role">textbox</property>
  </properties>
</spatialPresentation>
```

The relation between components and their properties can be formalized as follows:

**Definition 1** (*RUX-Method Components*)**.**

- Let RC be the set of components of the RUX-Method Concrete UI,
- Let P be the set of properties of Concrete UI components,
- Let M be the set of property-component relationships, $M \subseteq P \times RC$.
- A component $rc \in RC$ is a tuple $(ir, P')$ so that $\forall p \in P', \exists (p, rc) \in M$. □

### 3.2. Workflow specification in the RUX-Method

The RUX-Method expresses active interaction and temporal behaviors using Event-Condition-Action (ECA) rules. Each ECA rule consists of an event, defined in the Temporal or in the Interaction Presentation, and a Condition-Action-Tree (CAT), defined by a workflow in the behavior model. Conditions must be boolean expressions. Actions can be (1) calls to widgets methods (e.g., setcolor); (2) calls to the business logic components of the underlying model (e.g., a synchronous form POST); (3) call to temporal methods (e.g., stop); (4) an OCL expression.

The RUX-Method behavior workflow specification is based on BPMN. Although BPMN has been mainly conceived for business processes, it is flexible enough, as stated in Brambilla et al. (2005), for RIA behavior requirements. The main differences introduced by the RUX-Method to the BPMN notation are (1) a semantic match between actions (from ECA) and BPMN Activities notation and (2) typed activities as an extension to BPMN in order to differentiate the type of RUX-Method actions explained before (e.g., calls to business logic or temporal presentation methods). The latter is also used to provide the right selection of values in RUX-Tool as well as a faster code generation.

[7] http://www.ruxproject.org/nativecomp.pdf.

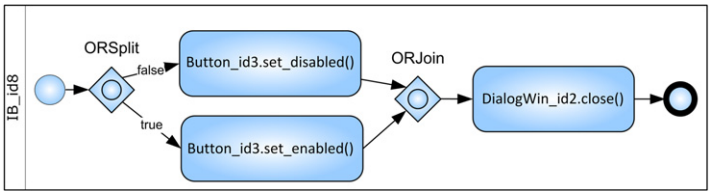


**Fig. 2.** Simple workflow of activities in BPMN.

Fig. 2 shows an excerpt of a behavior workflow. In the example, when a condition is satisfied (e.g. the *AutoCompleteTC* shown in Section 3.1 is filled) then a certain button is enabled; otherwise, the button is disabled.

## 4. Introducing accessibility in the RUX-Method

In order to introduce accessibility issues in the RUX-Method some changes have been done in the component definitions and the Transformation Rules between components placed at different levels. Next, we see how the RUX-Method incorporates the WAI-ARIA concepts identified in Section 2.1.

(1) *tabindex* was incorporated in the RUX-Method as a read-only property of the *BaseComponent* without specific values. The *tabindex* semantic is available in many RIA UI technologies where it is typically a property with positive integer values. However, WAI-ARIA introduces a new semantic to the term, allowing the $-1$ value. Due to the fact that the RUX-Method Spatial Presentation is specified as a hierarchical tree of components where the sibling components can be sorted, tabindex values are generated for the final UI according to their position in the UI tree. The update we have carried out to adapt the *tabindex* at the Concrete Interface level to the WAI-ARIA semantic was changing the property allowing read/write with only two valid values (i.e., *enabled* and *disabled*). This approach lets us take advantage of the already available transformation rules while maintaining compatibility with non-AJAX final rendering technologies in our other existing code generators. When the property value is enabled, TR2 are applied independently of the final RIA UI technology generating values between 1 and *N*. When the property is *disabled*, and only when the final UI technology is AJAX, the value generated is $-1$. For other RIA languages e.g., OpenLaszlo not allowing the definition of negative *tabindex* values, the *disabled* value in these cases is mapped to 0.

(2) The live region properties are introduced again in the *BaseComponent*. The problem is identifying the live regions in the RUX-Method and deciding when and how to change their properties. Regarding the first problem, we assume in the RUX-Method that a component is a live region if at least one of the following conditions is accomplished: (a) the component contains media components that retrieve information from the business logic specified in the underlying web model; (b) The component contains other components and is able to show, hide or change part of the UI. (e.g., a tab panel where only one "child" is shown at a time). Regarding the second problem, in both cases (a and b) the *aria-live* attribute is set to *polite*, *aria-relevant* is set to *all* and *aria-atomic* is set to *true*. For *aria-busy*, using Transformation Rules 1, the RUX-Method automatically creates conditions and actions in the Interaction Presentation to set the right value of the attribute at runtime.

(3) the RUX-Method includes the *role* property (explained in Section 4.1).

(4) the RUX-Method components already include states. They have at least one state and can declare others when required. WAI-ARIA attributes of widgets are equivalent to properties of RUX-Method components.

(5) The solution is based on roles as stated in 2.1. For the specific case of navigation sections the value of the property is automatically set by the TR1 due to the fact that these navigational links are extracted from the underlying web model.

The introduction of the role concept in the RUX-Method has two main pillars: ontoRUX, an extension of the WAI-ARIA ontology, and editRUX, an editor to enrich the RUX-Method Component Library with accessibility attributes which has been adapted from the SAW project.

```
<owl:Class rdf:ID="TextBox">
 <rdfs:subClassOf>
  <owl:Class> <owl:unionOf rdf:parseType="Collection"> <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:hasValue rdf:datatype="http://www.w3.org/2005/07/aaa#hidden">true</owl:hasValue>
        </owl:Restriction>
        <owl:Restriction>
     <owl:hasValue rdf:datatype="http://www.w3.org/2005/07/aaa#autocomplete">false</owl:hasValue>
        </owl:Restriction>
      </owl:intersectionOf>
     </owl:Class> <owl:Restriction>
      <owl:hasValue rdf:datatype="http://www.w3.org/2005/07/aaa#hidden">false</owl:hasValue>
     </owl:Restriction>      </owl:unionOf>          </owl:Class>   </rdfs:subClassOf>  </owl:Class>
```

**Fig. 3.** Example of restriction existing in ontoRUX.

### 4.1. ontoRUX

The original ontology provided by WAI-ARIA does not include definitions to check for possible inconsistencies in the accessibility properties of widgets. This is the motivation for ontoRUX which is an ontology that extends the WAI-ARIA ontology by adding restrictions in this taxonomy to keep the consistency in the property values of each instance of RUX component. Each class of ontoRUX is identified with a role of WAI-ARIA. For example, the *textbox* class in ontoRUX contains attributes describing its accessibility: *autocomplete*, *multiline*, *readonly* and *required*. It also contains other inherited attributes such as *hidden*, which are specified in the *roletype* class, the highest one in the hierarchy defined by WAI-ARIA ontology. In addition to this representation, ontoRUX contains several simple restrictions in OWL over the values in the attributes to keep the semantic consistency of this role. For example, Fig. 3 shows a restriction to avoid *textbox* instances to have simultaneously the *true* value in the attributes *required* and *hidden*.

The relation between ontoRUX roles (represented as classes in the ontology) and their attributes can be formalized as follows:

**Definition 2** (*ontoRUX Roles*). • Let R be the set of roles defined by ontoRUX,
- Let A be the set of accessibility attributes,
- Let P be the set of role-attribute relationships, with $P \subseteq R \times A$,
- Let C be the set of attribute restrictions, with $C \subseteq A \times A$.
- A role $r \in R$ is a tuple $(io, A', C')$ so that $\forall a \in A', \exists (r, a) \in P$ and $\forall c \in C', \exists a1, a2 \in A'$ so that $(a1, a2) \in C$. □

It is convenient to make clear that these roles (R) are represented in ontoRUX with the (*io*) identification tag (in OWL: *<owl:Class rdf:ID="identification name">*), and its ($a \in A'$) accessibility attributes (in OWL: *<role:supportedState rdf:resource="type of accessibility"/>*).

### 4.2. Combining ontoRUX and the RUX-Method

The enrichment of the RUX-Method components with accessibility attributes is done through editRUX, which is a new software component integrated in RUX-Tool. editRUX takes as entries the RUX-Method Component Library and ontoRUX, which is situated in a common repository.

The join point between components in the Component Library of the RUX-Method and components in ontoRUX is the *role* property. Each component of the Component Library of the RUX-Method is analyzed by editRUX looking for its role property. Once the *role* property is identified, the set of accessibility attributes and states for that role is searched in the ontology. Then, the set of properties of that component is augmented with those properties specified for that role in ontoRUX. This process is marked with a shadowed rectangle in Fig. 4 and will be automatically repeated whenever a change takes place in the Component Library (e.g. a new component added) or in ontoRUX (e.g. new elements or attributes added by WAI-ARIA).

The enriched Component Library serves as an entry for instantiating the components along the normal RUX-Method workflow (marked *RUX-Method UI design* in Fig. 4). Two validation processes

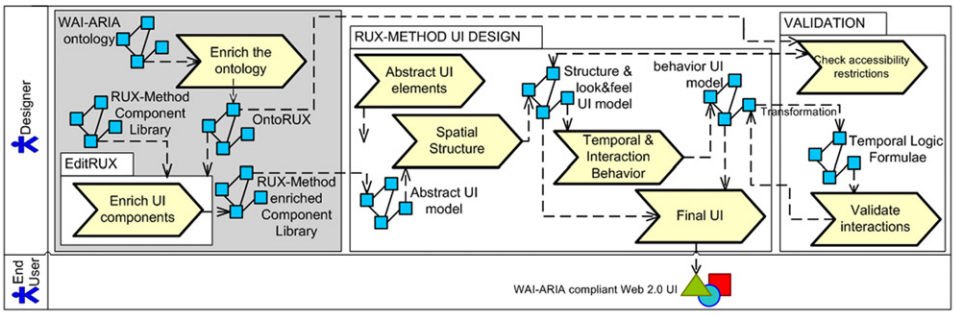


**Fig. 4.** Diagram of the process and related models of the approach.

have been added (marked *validation* in Fig. 4). On the one hand, the process that validates the restrictions established by ontoRUX (from the structure and look&feel UI model) and, on the other hand, the process validating the behaviors to ensure that the user will be notified of whatever change in the UI (from the behavior UI model). For the purpose of this paper we concentrate on this last process in Section 4.3.

The relation between RUX-Method components and ontoRUX roles can be formalized as follows:

**Definition 3** (*Accessible RUX-Method Components*)**.**

- Let ARC be the set of components of the RUX-Method Concrete UI enriched with accessibility properties.
- Let f: A $\rightarrow$ P be the injective application defined between the set of accessibility attributes of the ontoRUX ontology (A) and the set of RUX-Method component properties (P).
- A component arc $\in$ ARC is a tuple (ir, K) so that $\forall$ k $\in$ K,
  - . $\exists$c $\in$ C, c is a tuple (io, A′), io = ir, A′ $\subseteq$ A $\Longrightarrow$ f(A′) $\subseteq$ K.
  - . $\exists$rc $\in$ ARC, rc is a tuple (ir′, P′), ir′ = ir, P′ $\subseteq$ P $\Longrightarrow$ P′ $\subseteq$ K. $\square$

Next, we show the *AutoCompleteTC* component in the Final UI after being enriched according to the WAI-ARIA *textbox* role. In the example, we use for the implementation an auto-complete AJAX component which needs to include a library and a specific script code to work.

```
<input id="FTMI1"          type="text"       name="FTMI1"
  class="AutoCompleteTC" role="TextBox"    value=""
  autocomplete="inline"  readonly="false" hidden="false"
  multiline="false"      required="true"
  describedby="It is a textbox to fill the user address"/>

<script type="text/javascript">
  new Ajax.Autocompleter('FTMI1','update', page1.do?
     source=DBConn1_store.data.items[0].data.mytext',
     {tokens:',',});
</script>
```

When generating the code (i.e., TR2) those properties with static values are fixed as HTML/CSS properties, while for those ones that get values from the underlying business logic it is generated scripting code (e.g. by means of JQuery in AJAX). However, these generation rules are broken for accessibility properties that are always scripting code because, otherwise, the generated code with WAI-ARIA properties would not be valid XHTML, e.g., $('CTMI1').setProperty('aria-required', 'true').

### *4.3. Workflow validation in the RUX-Method*

Component properties can be changed at run-time due to user interactions or predefined temporal behaviors. Many of these changes may affect the UI and, consequently, they may affect accessibility

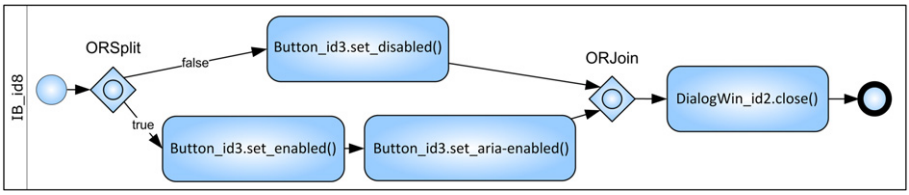


**Fig. 5.** Extended workflow of behaviors with actions related to accessibility.

properties. These changes must be reflected in the BPMN behavior workflow. Before generating the Final UI, it is necessary to ensure that every change in a component leading to a change in the UI also leads to a change in a WAI-ARIA property, so a disabled user can be notified of whatever change. Here, our approach is using a validation process by means of a symbolic computation technique.

Although there exist tools to simulate BPMN workflows, there is a lack of mechanisms to perform BPMN validations. However, an interesting approach to BPMN formalization using LTL formulae (Emerson, 1990) is presented in Brambilla et al. (2005). The RUX-Method takes advantage of this approach by generating LTL formulae from the BPMN diagrams. The RUX-Method works with an XML representation of the BMPN workflows. Using XSLT it obtains LTL formulae. In order to ease the validation operations, the XSLT allows enriching the temporal modal operators of LTL with variables, constants and Boolean functions to obtain First Order Linear Temporal Logic (FO-LTL) (Emerson, 1990). These formulae can be used by different kinds of tools (i.e. Spin[8]) allowing in this way to analyze the behavior and properties of the designed components.

To translate the BPMN elements and obtain a decidable and analyzable model in PSPACE (Deutsch et al., 2004) it is necessary to delimit the range of values of those variables used in the model. It is necessary also to make some considerations (Brambilla et al., 2005) such as, for example, that each activity represented in BPMN can get two possible states ("a" –activated-, "c"-completed) and before reaching the *c* state it must be at the *a* state.

With the aim of illustrating the validation process, Fig. 5 extends Fig. 2 with accessibility actions: when a button is enabled (*true* flow in the figure), then a WAI-ARIA property is changed to notify e.g., an AT about the change in the UI.

Next we show the generated LTL formula from the previous diagram, where **F** stands for the Finally operator and **B** for the Before operator.

$$
\begin{gathered}
(\mathbf{F}\ \text{ORSplit1}) \\
\bigcap(\text{ORSplit}\ \mathbf{B}\ \neg(\mathbf{F}\ \text{Button\_id3.set\_disabled().a} \bigcup \mathbf{F}\ \text{Button\_id3.set\_enabled().a})) \\
\bigcap(\text{Button\_id3.set\_enabled().c}\ \mathbf{B}\ \neg\ (\text{Button\_id3.set\_aria-enabled().a})) \\
\bigcap((\text{Button\_id3.set\_disabled().c} \bigcup \text{Button\_id3.set\_aria-enabled().c})\ \mathbf{B}\ \neg\ (\text{ORJoin1})) \\
\bigcap((\text{ORJoin1})\ \mathbf{B}\ \neg\ (\text{DialogWin\_id2.close().a})) \bigcap (\text{DialogWin\_id2.close().c})
\end{gathered}
$$

The LTL representation of the components allows analyzing the usability of the model, considering safety, liveness and fairness properties. Thus, it is possible to study whether the accessibility properties are integrated and instantiated properly, how the underlying properties of the system are affected, the interactions between components, and in general if the evolution in the state of the components and the system is satisfactory.

Once the validation is performed with success, the final application is generated and it can be rendered using whatever combination of browser and WAI-ARIA compliant AT to make the interaction with RIA UIs accessible for people with special needs. Currently, there are several browsers and assistive technologies that give support to the WAI-ARIA draft, such as the latest versions of Firefox, IE, Opera or Jaws.

8 http://spinroot.com/spin/whatispin.html.

## 5. Conclusions and related work

This paper has introduced the combination of two previous works, the RUX-Method and SAW, to obtain accessible Rich Internet Applications. The result is a process that allows enriching already developed Model-Based Web applications with RIA features and accessibility issues. The main pillar of this combination is ontoRUX, an ontology based on WAI-ARIA ontology defining the attributes that must contain those widgets introduced by RIA technologies in order to be interpreted by the combination of accessibility APIs and assistive technologies. OntoRUX also specifies the restrictions to keep the consistency in the values of these attributes.

Talking about related works is not an easy task due to several reasons:

(1) Currently, accessibility issues are not being taken into consideration by Web models (Rossi et al., 2008). Although some of these proposals have advanced approaches to consider RIA features, these ones are mainly related to aspects different from the UI design (logic or data distribution, synchronization, etc) and there is no known roadmap to include accessibility issues.
(2) Those works addressing accessibility have a limited coverage and fall in one of the following fields:
   (a) Works that are focused on the evaluation of the accessibility degree of Web pages such it is the case of Taw,[9] Hera,[10] or Wave.[11] These tools can be considered as post-implementation tools. A detailed explanation of tool support for accessibility assessment can be found in Xiong and Winckler (2008). In this context editRUX can be seen as an evaluation tool but it goes a step further allowing the annotation of those inaccessible elements.
   (b) Works that are focused only on the UI design (Vanderdonckt et al., 2004), paying no attention to the connection with other models providing data, business logic or communication issues which are very relevant in RIAs. The RUX-Method also concentrates on the UI design, but provides appropriate mechanisms to connect to underlying Web models so applications with inaccessible Web 1.0 UIs developed with these models can be converted into accessible Web 2.0 UIs.
   (c) Works that indicate guidelines to include accessibility at the code level as it is the case of WCAG or many other papers (Kern, 2008) or web pages that one can find on the Web to make AJAX more accessible. In the context of the RUX-Method these guidelines have been taken into account when generating the final code.
   (d) Works that are mainly focused on code such as AXSJAX. In this sense, the RUX-Method goes a step further working at two different levels: at the code level when injecting WAI-ARIA properties using JQuery in TR2, and at the model level when enriching the Component Library.

It must be understood that WAI-ARIA cannot be considered the panacea for obtaining accessible RIAs. Designers must consider some guidelines when building the UI such as those presented in Hailpern et al. (2009). Other work presenting guidelines is Xiong et al. (2008). In this case an ontology approach for specifying guidelines is presented. However, we preferred to work around WAI-ARIA in order to keep closer to this W3C standard.

Although the obtained results are promising, we have only considered simplified application scenarios. Current work includes both, developing complex WAI-ARIA compliant RIAs and testing the results with a broad spectrum of visually impaired users.

The BPMN expressive power is limited and, so, not all the LTL properties can be expressed (for example, BPMN cannot express the X-Next operator or negation). To increment its expressive power, the work in Brambilla et al. (2005) proposes BPMN extensions with primitives that allow the integration of properties that can be described with temporal logic. However, these primitives have not been included in the RUX-Method yet.

9 http://www.tawdis.net.
10 http://www.sidar.org/hera/index.php.
11 http://wave.webaim.org.

## Acknowledgement

The authors would like to thank the reviewers for their insightful and accurate comments.

## References

Borodin, Y., Bigham, J.P., Raman, R., Ramakrishnan, I.V., 2008. What's new?: making web page updates accessible. In: Assets'08: Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 145–152.

Brambilla, M., Deutsch, A., Sui, L., Vianu, V., 2005. The role of visual tools in a web application design and verification framework: a visual notation for LTL formulae. In: Lowe, D., Gaedke, M. (Eds.), ICWE. In: Lecture Notes in Computer Science, vol. 3579. Springer, pp. 557–568.

Chen, C.L., Raman, T.V., 2008. Axsjax: a talking translation bot using Google im: bringing web-2.0 applications to life. In: W4A'08: Proceedings of the 2008 International Cross-Disciplinary Conference on Web Accessibility, W4A. ACM, New York, NY, USA, pp. 54–56.

Deutsch, A., Sui, L., Vianu, V., 2004. Specification and verification of data-driven web services. In: PODS'04: Proceedings of the Twenty-Third ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems. ACM, New York, NY, USA, pp. 71–82.

Emerson, E.A., 1990. Temporal and modal logic. pp. 995–1072.

Hailpern, J., Guarino-Reid, L., Boardman, R., Annam, S., 2009. Web 2.0: blind to an accessible new world. In: WWW'09: Proceedings of the 18th International Conference on World Wide Web. ACM, New York, NY, USA, pp. 821–830.

Kern, W., 2008. Web 2.0 — end of accessibility? analysis of most common problems with web 2.0 based applications regarding web accessibility. International Journal of Public Information Systems 2, 131–154.

Linaje, M., Preciado, J.C., Morales-Chaparro, R., Sánchez-Figueroa, F., 2008. On the implementation of multiplatform RIA user interface components. In: IWWOST. pp. 44–49.

Linaje, M., Preciado, J.C., Sánchez-Figueroa, F., 2007. Engineering rich Internet application user interfaces over legacy web models. IEEE Internet Computing 11 (6), 53–59.

Lunn, D., Harper, S., Bechhofer, S., 2009. Combining sadie and axsjax to improve the accessibility of web content. In: W4A'09: Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibility, W4A. ACM, New York, NY, USA, pp. 75–78.

Rossi, G., Pastor, O., Schwabe, D., Olsina, L. (Eds.), 2008. Web Engineering: Modelling and Implementing Web Applications. In: Human–Computer Interaction, Springer, London.

Sánchez-Figueroa, F., Lozano-Tello, A., González-Rodríguez, J., Macías-García, M., 2007. Saw: a set of integrated tools for making the web accessible to visually impaired users. European Journal for the Informatics Professional, UPGRADE VIII (2), 67–71.

Stringer, E.C., Yesilada, Y., Harper, S., 2007. Experiments towards web 2.0 accessibility. In: HT'07: Proceedings of the Eighteenth Conference on Hypertext and Hypermedia. ACM, New York, NY, USA, pp. 33–34.

Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D., Florins, M., 2004. Usixml: a user interface description language for specifying multimodal user interfaces. In: Proc. of W3C Workshop on Multimodal Interaction, Sophia Antipolis, pp. 19–20.

Xiong, J., Farenc, C., Winckler, M., 2008. Towards an ontology-based approach for dealing with web guidelines. In: WISE'08: Proceedings of the 2008 International Workshops on Web Information Systems Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 132–141.

Xiong, J., Winckler, M., 2008. An investigation of tool support for accessibility assessment throughout the development process of web sites. Journal of Web Engineering 7 (4), 281–298.