Semantic Data Driven Interfaces for Web Applications

Vagner Nascimento and Daniel Schwabe

Department of Informatics, PUC-Rio,
Rua Marques de Sao Vicente, 225. Rio de Janeiro, RJ 22453-900, Brazil
{vnascimento,dschwabe}@inf.puc-rio.br

Abstract. Modern day interfaces must deal with a large number of heterogeneity factors, such as varying user profiles and runtime hardware and software platforms. These conditions require interfaces that can adapt to the changes in the <user, platform, environment> triad. The Model-Based User Interface approach has been proposed as a way to deal with these requirements. In this paper we present a data-driven, rule-based interface definition model capable of taking into account the semantics of the data it is manipulating, especially in the case of Linked Data. An implementation architecture based on the Synth environment supporting this model is presented.

Keywords: SHDM, HCI, Interface, Adaptation, Semantic Web, Data-driven design.

1 Introduction

The design and implementation of the interface component of applications, and in particular Web applications, consumes over 50% of the development effort, as first reported by, Myers and Rosson in the nineties [11]. Since then, their figures have surely increased, due to the evolution of the computing platforms, the advent of the Internet and the Web, and the now popular gestural and vocal interface modalities. Sources of heterogeneity affecting application development include:

- Different computing platforms – desktops, laptops, tablets, smartphones, embedded devices - affording a variety of interaction modalities – typing, voice, motion sensing, (multi)touch - and diverse input/output capabilities - keyboard, mouse, (multi)touch sensitive surfaces, motion sensors, cameras, even head-mounted displays/cameras;
- Multiple, often dynamically varying contexts of use, be it at a desktop with a wired network or a smartphone or Google Glass-like device on the go, wirelessly connected in a variety of underlying network infrastructures. These contexts also includ diverse working environments, that may have high degree of noise, and sometimes restricted bandwidth;
- Multiple, ever evolving set of tasks that must be supported, derived from an increasing number of different workflows that users adopt and must be supported by the application;

F. Daniel, P. Dolog, and Q. Li (Eds.): ICWE 2013, LNCS 7977, pp. 22–36, 2013.

- Highly diverse types and profiles of end users, ranging from very novice to experts, being from many different cultures and speaking a multitude of languages,

Not only these sources of heterogeneity exist, but often the context of use, i.e., each component of the triad <user, platform, environment> (the context) changes dynamically while the application is being used, which calls for so-called Plastic UIs [3], capable of adapting while preserving the "user experience" while the user is engaged with the application.

The Model-Based User Interface (MBUI) development approach has been used to address these challenges and maintain or decrease the level of effort necessary to develop applications, and more specifically, user interfaces, under these conditions.

The Cameleon Reference Model is a current reference framework for User Interfaces gaining adoption [2], the result of several years of research of a major European research project, which proposes four abstraction levels for modeling UIs: Task and Domain, Abstract Interface, Concrete Interface, Final User Interface.

The Domain model describes the domains of the application, and the Task model describes the sequence of steps needed to perform the tasks (with respect to interactions with the User Interface).

The Abstract Interface model describes the composition of interface units in an implementation and modality independent way.

The Concrete Interface model describes the interface in terms of platform-dependent widgets, but still modality- and implementation language independent.

The Final User Interface is the actual running code that the end user accesses when interacting with the application.

A more recent trend has been the dissemination of the Semantic Web, and the availability of data sources expressed in its formalisms – RDF, RDFS, OWL, in particular the Linked Data Initiative (LOD)[1], and the emergence of Linked Data Applications (LDAs for short), that access, enrich and manipulate linked data. There are some proposals of development environments or frameworks for supporting the development of LDAs, such as CubicWeb[2], the LOD2 Stack[3], and the Open Semantic Framework[4]. In addition, semantic wiki-based environment such as Ontowiki[5], Kiwi[6], and Semantic Media Wiki[7] have also been used as platforms for application development over Linked Data.

While useful, they do not present a set of integrated models that allow the specification of an LDA, and the synthesis of its running code from these models. Therefore, much of the application semantics, in its various aspects, remains represented only in the running implementation code.

---

[1] http://linkeddata.org
[2] http://www.cubicweb.org
[3] http://lod2.eu/WikiArticle/TechnologyStack.html
[4] http://openstructs.org/open-semantic-framework
[5] http://ontowiki.net/Projects/OntoWiki
[6] http://www.kiwi-project.eu
[7] http://www.semantic-mediawiki.org/wiki/Semantic_MediaWiki

We have been working in the past years in the Semantic Hypermedia Design Method (SHDM) [6] and its implementation environment Synth [1], which aims to allow Model-Based development of Linked Data Applications. While SHDM includes a proposed Abstract Interface Model, it lacks more refined models capable of dealing with the complexities of UIs as outlined above.

In this paper we present a new set of User Interface models and its implementation architecture similar to the Cameleon Framework proposal, addressing some of the challenges outlined earlier.

We present our approach in this paper as follows. After describing the example we are going to use through the paper in Section 2, we present our approach for interface modeling in Section 3. We discuss the implementation in Section 4. Section 5 presents the related work and with Section 6 we draw some conclusions and discuss future work.

2 Running Example

To help illustrate the concepts discussed in the paper, we use a running example of a fictitious online hotel-booking site. Suppose the user navigated to a given hotel's page, but has not yet entered the date, then the page should include fields to allow her/him to enter the desired dates, as shown in **Fig. 1** and **Fig. 2**.

When the dates have been informed, the application must show the rates for each type of room, their availability, and a warning is there is low availability for a certain type of room.

Notice that these conditions depend both on Domain Model information, and on the interaction state. The actual screen layout and interaction options depend also on the device; **Fig. 1** and **Fig. 2** show here the interface meant for desktop computers.

Fig. 3 shows the same application when accessed from a mobile device, with a different layout and different interaction capabilities (e.g., scrolling through swiping across the screen).

Fig. 1. Example hotel details page, with fields to inform check-in and checkout dates

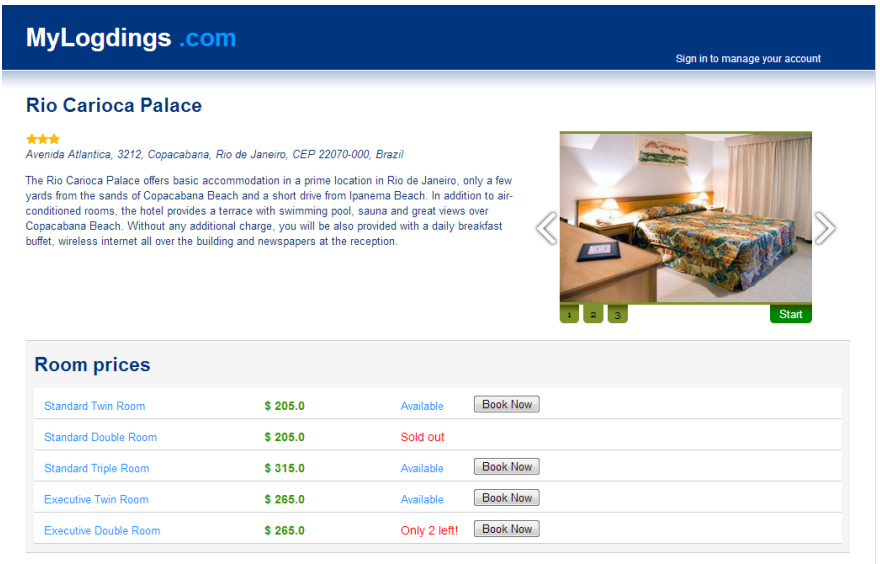

Fig. 2. – Details of available hotel rooms if the dates have been provided

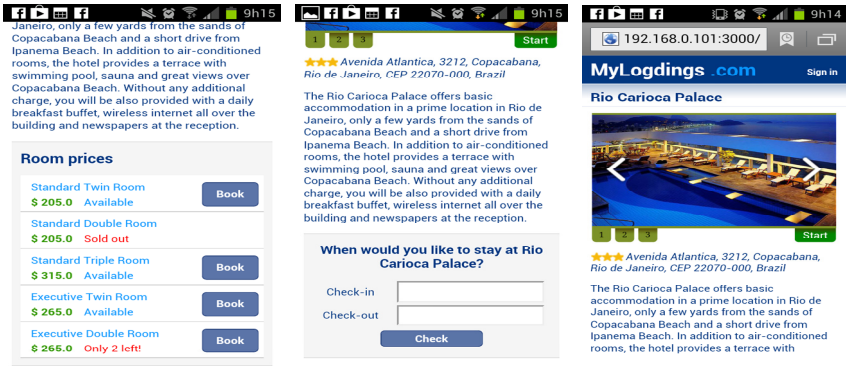

Fig. 3. – Mobile device version of the hotel-booking example interfaces

3 A Semantic Interface Model

In this section we present the new set of models for specifying interface in SHDM[8]. As mentioned earlier, SHDM follows the basic abstraction levels of the Cameleon Reference Model. The Domain Model, in SHDM is simply a set of RDF triples, which form a graph, and may include RDFS or OWL definitions. It is often the case

[8] A video illustrating the use of these models in Synth is available in `http:://www.tecweb.inf.puc-rio.br/synth`

that there does not exist any schema definitions in the Domain Model, only instances of resources representing information items.

The Abstract Interface Model [14] focuses on the roles played by each interface widget in the information exchange between the application and the outside world, including the user. It is abstract in the sense that it does not capture the look and feel, or any information dependent on the runtime environment. The Concrete Interface model is responsible for the latter.

Summarizing the Abstract Interface meta-model, an abstract interface is a composition of abstract interface elements (widgets). These in turn can be an `ElementExhibitor`, which is able to show values; an `IndefiniteVariable`, which is able to capture an arbitrary input string; a `DefinedVariable`, which is able to capture input values (one or several) from a known set of alternatives; and a `SimpleActivator`, which is able to react to an external event and signal it to the application.

Consider the interfaces shown in **Fig. 1**-**Fig. 3**. From them we can see that a hotel page has

- A header with a title and an anchor to the login operation;
- Hotel data, including name, address, category, description;
- A set of hotel images;
- A table of room types and respective rates, their availability, and an anchor to book it;
- A form to input check-in and checkout dates.

The corresponding abstract interface describing this is (as a nested list of attribute-value pairs)

```
{name: "main_page", widget_type: "AbstractInterface", children:[
  {name: "header", widget_type: "CompositeInterfaceElement",
children: [
    {name: "title", widget_type: "ElementExhibitor"},
    {name: "account_anchor", widget_type: "SimpleActivator"},
   ] },
  {name: "content", widget_type: "CompositeInterfaceElement",
children:[
    {name: "hotel_name", widget_type: "ElementExhibitor"},
    {name: "hotel_images", widget_type:
"CompositeInterfaceElement", repeatable: true, children: [
      {name: "hotel_image", widget_type: "ElementExhibitor"}
      ]},
    {name: "hotel_category", widget_type: "ElementExhibitor"},
    {name: "hotel_address", widget_type: "ElementExhibitor"},
    {name: "hotel_description", widget_type:
"ElementExhibitor"},
    {name: "rates", widget_type: "CompositeInterfaceElement",
children: [
      {name: "rates_title", widget_type: "ElementExhibitor"},
      {name: "rates_by_room", widget_type:
"CompositeInterfaceElement", repeatable: true,
```

```
        children: [
          {name: "room", widget_type:
"CompositeInterfaceElement", children: [
            {name: "room_type", widget_type:
"ElementExhibitor"},
            {name: "price", widget_type: "ElementExhibitor"},
            {name: "availability", widget_type:
"ElementExhibitor"},
            {name: "book", widget_type: "SimpleActivator"}
         ]},
        ]},
      ]},
    {name: "search_rates", widget_type:
"CompositeInterfaceElement", children: [
      {name: "search_rates_title", widget_type:
"ElementExhibitor"},
      {name: "label_checkin", widget_type: "ElementExhibitor"},
      {name: "checkin", widget_type: "IndefiniteVariable"},
      {name: "label_checkout", widget_type: "ElementExhibitor"},
      {name: "checkout", widget_type: "IndefiniteVariable"},
      {name: "search", widget_type: "SimpleActivator"}

      ]}
    ]}
  ]}
```

Fig. 4. - Abstract Interface specification of the Interfaces in **Fig. 1**-**Fig. 3**

Notice that this Abstract Interface represents both interfaces; each specific one can be seen as a special case of this one, where some elements have been omitted. The Abstract Interface also adds the widget types, indicating their role in the information flow.

A mapping specification made by the designer determines how each abstract widget will be mapped onto one or more Concrete Interface elements, and onto which Operations. The latter are the primitives in SHDM used to specify the business logic i.e., the application behavior to achieve the desired tasks.

Here we start introducing the new features in the existing model. Previously, the designer would determine, for each operation, which abstract interface would be used to exhibit its results. Furthermore, the composition of widgets in each abstract interface was specified statically at design time, the same being true for its mapping to concrete interfaces.

The new model instead uses rules to determine each of these aspects. Thus, instead of statically defining which abstract interface should be used, how that interface is composed, and how it is mapped onto the concrete interface, the designer now establishes *rules*, which, in a model- (and data-) driven fashion will assemble the final user interface that will be used. **Fig. 5** shows how the Interface Models are related to each other, and how the actual interface is defined.

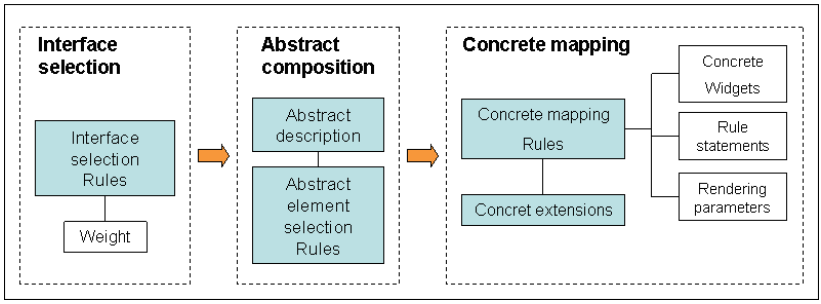

Fig. 5. – Relation between Interface Models in SHDM

The first step is the selection of the abstract interface, determined by its own set of rules. The result of executing these rules is a ranked list of candidate Abstract Interfaces, based on a weighting function defined by the UI designer.

The highest-ranking Abstract Interface is then chosen. Its own composition is again determined by executing another set of rules, which may include or exclude widgets from the initial base Abstract Interface composition defined by the designer.

Next, a third set of rules is executed to determine how each Abstract Interface widget will be mapped onto concrete interface widgets, and in some cases also extend the concrete widget compositions to allow interaction between them.

This rule-driven approach has several advantages:

1. It allows taking into account actual runtime data and context information in determining which interface should be used. Since the rules can refer to actual input data to be exhibited through it, as well as to the Domain Model, it is fair to say that the interface definition is now Semantic, in the sense of being aware of the data types and values of the data it is exposing;
2. It allows adapting the interface to both the user and to the execution environment, allowing a user experience that is in tune with the user's device and environment capabilities. Once again, such rules may take into account the semantics of the user or context model to alter the concrete interface.
3. It becomes a design choice whether the adaptation process will be run only at design time, or also during runtime. Running them during the application execution provides maximum flexibility, as the interface can change dynamically in reaction to several context changes, such as change of device, reduced bandwidth, loss of modality due to either circumstantial reasons (e.g., no visual access during driving) or due to hardware failure (e.g., display failure).

3.1 Rules and Interface Definition Parameters

Before going into more detail on how each part of the Interface Model is specified, it is useful to summarize the different types of information that are the input parameters for the definition.

- Rules follow the Condition-Action format. The conditions can reference
  - Any of the other models in SHDM, namely, Domain, Hypertextual Navigation, and Operations. For instance, it can test the type and value of a data item, or whether the element being exhibited is a hypertextual link;

  - Hypertextual parameters received in an http request;
  - Browser header information, including browser, platform, operating system, etc.
  - Environment variables, e.g., date and time of day, location
- Mapping specifications are a different type of rule, which use data both to establish the concrete interface to be activated, and to pass rendering parameters as needed. These include hypertextual navigation information, including sets of values to be iterated over.

All this information is converted into <object, property, value> triples which are input to the rules facts database. The pre-condition of each rule simply tests the presence or absence of a triple pattern in the facts database.

When an Operation (a behavior specification in SHDM) activates the Interface Engine to render its results, it also passes parameters needed for the rendition. Such parameters typically include the Domain Model data values and any input parameters it has received itself.

We next discuss each type of rule, illustrating it with the running example.

3.2 Abstract Interface Selection Rules

The first step in defining the Interface is establishing the selection rules for the Abstract Interface. The pre-condition in these rules define when each Abstract Interface is applicable, allowing, for instance, to

- Select the interface only if the user is logged in;
- Select the interface only if the application is being accessed from a mobile device;
- Select the interface only for certain types of data passed as input during runtime. Notice that this is often necessary if one wants to deal with "raw" data in RDF, which may not have any schema or vocabulary information associated with it.

In our example, the Abstract Interface selection rules are

```
set{
  has "params", "action", "hotel"
  has "params", "id", :_
}
```

The first line in the set tests whether we are exhibiting a `hotel` page; the second tests whether a specific hotel (i.e., `id` has some value) was passed as a parameter.

3.3 Abstract Interface Element Selection Rules

The Abstract Interface is a composition of elements. Each element may have rules associated to it, which determine if that element will be included in the final Abstract Interface composition or not.

Consider the rates element in the Abstract Interface shown in **Fig. 4**. It should be shown only if the check-in and checkout dates have been defined; conversely, the input fields for those dates (the search_rates element) should be shown if they have not been defined. The following rules capture this. The neg condition is the same as not has.

```
set "rates" do
  has "params", "checkin", :_
  has "params", "checkout", :_
end
```

```
set "search_rates" do
  neg "params", "checkin", :_
  neg "params", "checkout", :_
end
```

3.4 Concrete Interface Mapping Rules

For each Abstract Interface widget, there is a mapping rule that determines how it is mapped onto concrete widgets. Below we show some of the rules that map the Abstract Interface in **Fig. 4** onto the concrete interfaces of **Fig. 1**-**Fig. 3.**

Each rule starts with `maps-to`, includes the name of the abstract widget it applies to; the concrete widget to which it maps; parameters needed by the concrete widget; and a rule block delimited by `do-end` used to determine under which conditions the mapping is applicable. Rules are applied in order; once a rule has been applied to an element, other subsequent rules applying to that same element are ignored.

```
1. maps_to abstract: "main_page", concrete_widget: "HTMLPage" ,
   params: { title: "myLogdings.com - #{hotel[:name]}",
   include_css: "/stylesheets/hotel_mob.css" }do
2. has "user_agent", "mobile", true end
3. maps_to abstract: "main_page", concrete_widget: "HTMLPage" ,
   params: { title: "myLogdings.com - #{hotel[:name]}",
   include_css: "/stylesheets/hotel.css" }
# Header block
4. maps_to  abstract: "header", concrete_widget:
   "HTMLComposition"
5. maps_to  abstract: "title", concrete_widget: "HTMLHeading",
   params: { content: "MyLogdings" }
6. maps_to  abstract: "account_anchor", concrete_widget:
   "HTMLAnchor", params: { content: "Sign in to manage your
   account", url: "/signin" }
7. maps_to  abstract: "content", concrete_widget:
   "HTMLComposition"
# Hotel Data
8. maps_to  abstract: "hotel_name", concrete_widget:
   "HTMLHeading", params: { size: 2, content: hotel[:name] }
   # Images slider
9. maps_to  abstract: "hotel_images", concrete_widget:
   "JQueryAnythingSlider", params: { collection:
   hotel[:images], as: :hotel_image }
10. maps_to  abstract: "hotel_image", concrete_widget:
   "HTMLImage", params: { content: hotel_image }
…
# Rates
```

```
11. maps_to  abstract: "rates", concrete_widget:
    "HTMLComposition"
```

…

```
#== Availability
12. maps_to  abstract: "availability", concrete_widget:
    "HTMLSpan", params: {content: "Sold out", css_class:
    "highlight"}do
13. equal room[:status], 'sold-out' end
14. maps_to  abstract: "availability", concrete_widget:
    "HTMLSpan", params: { content:
    "Only #{room[:rooms_available]} left!", css_class:
    "highlight"}do
15. equal room[:status], "few-rooms" end
16. maps_to  abstract: "availability", concrete_widget:
    "HTMLSpan", params: {content: "Available", css_class: "col3"
    }
17. maps_to  abstract: "book", concrete_widget:
    "HTMLFormButton", params: {content: "Book", css_class:
    "col4"} do
18. diff room[:status], "sold-out"
19. has "user_agent", "mobile", true end
20. maps_to  abstract: "book", concrete_widget:
    "HTMLFormButton", params: {content: "Book Now", css_class:
    "col4"} do
21.   neg "user_agent", "mobile"
22.   diff room[:status], "sold-out" end
# Search rates
23. maps_to abstract: "search_rates", concrete_widget:
    "HTMLForm", params: {method: "get" }
24. maps_to  abstract: "search_rates_title", concrete_widget:
    "HTMLHeading", params: {size: 2, content: "When would you
    like to stay at #{hotel[:name]}?"}
25. maps_to  abstract: "label_checkin", concrete_widget:
    "HTMLLabel", params: {content: 'Check-in' }
26. maps_to  abstract: "checkin", concrete_widget:
    "JQueryDatePickerInput" , params: {date_format: "d M, y",
    min_date: 0 }
27. maps_to  abstract: "label_checkout", concrete_widget:
    "HTMLLabel", params: {content: 'Check-out' }
28. maps_to  abstract: "checkout", concrete_widget:
    "JQueryDatePickerInput" , params: {date_format: "d M, y",
    min_date: 0 }
29. maps_to  abstract: "search", concrete_widget:
    "HTMLFormButton", params: {content: "Check" }
```

Some concrete widgets, such as HTMLHeading, HTMLSpan, HTMLForm, etc… correspond directly to their counterparts in HTML. We make additional comments highlighting the interesting uses of the rules.

- Lines 1 and 3 show two possible mappings for the main page. The first is selected when the user agent is a mobile device, tested in line 2. Otherwise, the mapping in line 3 applies. This is how the proper choice for generating of the interfaces in **Fig. 1**- **Fig. 3** is made.
- The expression `#{hotel[:name]}` in line 1 retrieves the value of the "name" property of the hotel instance being shown;
- The expression `url: "/signin"` in line 6 generates a (REST) call to the `signing` Operation, defined in the Behavior Model (not shown);
- Line 9 shows the use of a Javascript component. `JQueryAnythingSlider`, capable of exhibiting a set of elements, including images. The actual set of elements is passed as a parameter, the result of the expression `collection: hotel[:images]` that retrieves from the Domain Model the set of image values associated with the hotel being exhibited. Lines 26 and 28 map the input form fields for the check-in and checkout dates to a library component, `JQueryDatePickerInput`.
- Line 12 shows a conditional element. If the value of the `room[:status]` property is `"sold out"`, this element (a warning text "Sold out") will be shown, with a CSS style `"highlight"`.
- Lines 14-15 show another conditional element. If the value of the `status` property of `room` is `"few-rooms"`, a highlighted warning showing the number of rooms left (`"Only #{room[:rooms_available]} left!"`) is shown; otherwise it is omitted.
- The `book` element defined in Line 17 is only included if there are rooms available, as specified in the condition in line 18. There are two different CSS styles used, one when the user agent is a mobile device (tested in line 19), the other when it is not (tested in line 21).

An interesting point is raised by the flexibility of the mapping rule language. Since any valid DSL expression (see [12] for a discussion on the embedded DSL offered by Synth) can be used in the test clause of the condition, we could have inserted the test for low availability in the rule itself, e.g., `{room[:rooms_available]} < 3}`. This, however, would imply including parts of the Business Logic in the interface, which is undesirable. Rather, this condition is actually implemented as an inference rule in the Domain Model, which concludes the fact `<"room", "status", "few-rooms">` from the number of rooms available, according to the application's Business Rules.

In addition to these mapping rules, it is sometimes necessary to define Extensions to the Concrete Interface Model to allow interactions between concrete widgets. A common example is when the value set to one widget must be used as an input to another widget.

Consider the check-in and checkout date widgets specified in lines 26 and 28. It would be user-friendlier (and semantically correct) that once the check-in date has been filled, the checkout date should be a date at least one day later. The extension shown below encapsulates this behavior:

```
extend nodes: ['checkin'], extension: 'JQueryCopyDateTo',
params: { target: "checkout", string_format: "d MMM, yy",
add_days: 1 }
```

Extensions are wrappers around Concrete Interface elements. Typically, they will call Domain model operations to determine Domain-dependent integrity constraints normally enforced by these communications between widgets.

3.5 Concrete Widgets Definitions

As seen from the examples in the mapping rules, concrete widgets are treated as software components outside the model itself; different concrete widgets should be defined for different runtime platforms. In this sense, we diverge from the Cameleon model, as Concrete Widgets are rendered directly to the Final User Interface.

A Concrete Widget should be self-contained, and capable of self-rendering based only on their input parameters. Any potential dependencies they may have with other widgets should be parameterized as well. For example, the `JQueryDatePickerInput` is capable of receiving an initial date, as used by the extension discussed above in the case of check-in and checkout dates.

Concrete Widgets are described in Manifest declarations, containing their name; version; description; list of compatible abstract widgets (i.e., abstract widgets that can be mapped to it); list of other widgets it depends on; list of parameter; and a text with examples of use.

4 Implementation Architecture

The conceptual architecture that integrates the models defined in Section 3 is show in **Fig. 6**.

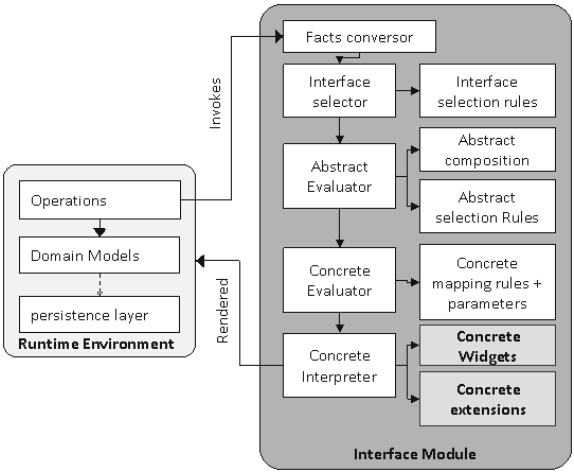

Fig. 6. – The conceptual implementation architecture for Interfaces

The Facts Convertor component is responsible for extracting the model definitions from the knowledge base, and converts them into facts - <object, property, value> triples - that will be used by the rules engine. The Interface Selector runs the Selection rules, returning a ranked list of interfaces. The Abstract Evaluator runs the composition rules, resulting in the actual Abstract Interface to be used; abstract widgets without associated rules are included by default. The Concrete Evaluator runs the mapping rules to generate the concrete interfaces, adding applicable extensions, and the results are interpreted using the concrete widget definitions to generate the final running interface.

The Concrete Interface Interpreter receives a composition tree of concrete widget specifications, including their parameters and extensions. It does a depth-first traversal of the tree, and for each node instantiates (i.e., generates the code) for the corresponding concrete widget.

Fig. 7 shows the actual sequence of events within the Interface Engine in Synth.

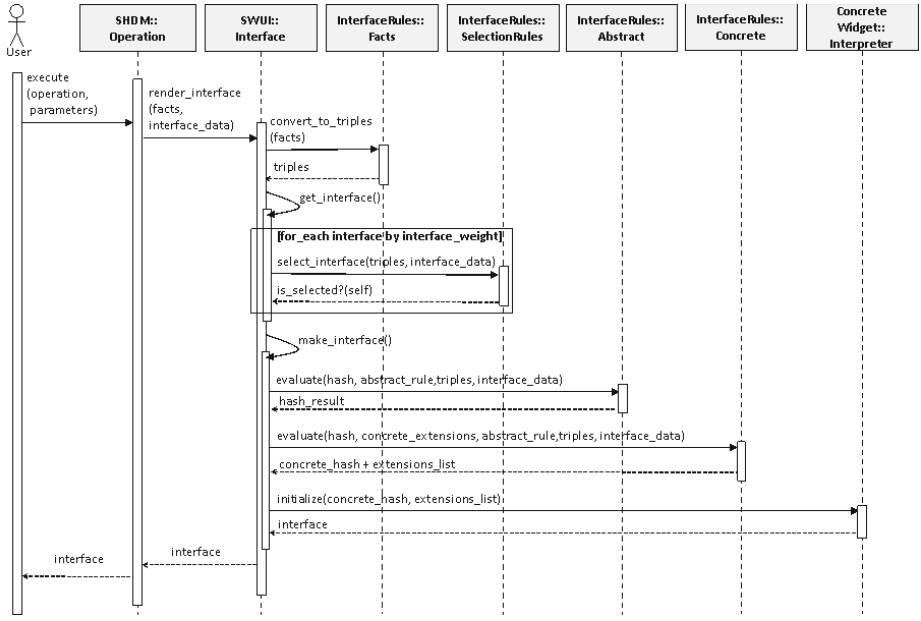

Fig. 7. – Sequence of events in the implementation of the Interface Engine in Synth

The Interface Engine is implemented in Ruby, as is the Synth environment. The rules engine used is *Wongi-Engine*[9], implementing the classical RETE algorithm.

5 Discussion and Conclusions

We have described a data- and model-driven rule based model and runtime architecture. It is data-driven since the actual interface is self-assembled as a result of

[9] `https://github.com/ulfurinn/wongi-engine`

the execution of the various rule-sets that use the instance data in the various models in SHDM (Domain, Hypertextual Navigation, Behavior) to determine the final interface. It is model-driven because all Synth models are available as data as well (as discussed in [1]). For example, a rule can determine the inclusion of an abstract widget if the data item being exhibited is of a certain type, and/or if it has a certain property, e.g., "it is of any Class that has a Discount property".

The work presented here is related to a very large number of models and approaches that have been proposed in the literature (see, for example, [10]); it would be beyond the scope of this paper to make a comparison with every one of them. Several of the Interface Models in SHDM, e.g., the Abstract Interface and the Concrete, have counterparts in the many proposed models, e.g., Maria [13], UsiXML [9], UIML [7], among many, as well as those in Hera [5], UWE [8] and WebML [4], differing mostly in the level of abstraction and on the underlying formalism (e.g., XML vs RDF). Each has advantages and disadvantages, a discussion of which would require another paper altogether. A similar observation can be made regarding the use of rules (e.g. [15], the difference still remaining in the underlying models.

The major distinguishing original contribution is the use of data- and model-driven rules integrated seamlessly with the various other models within the SHDM approach, directly supported by an implementation environment. Our approach leads to explicating design decisions associated to the various levels of abstraction, as they become explicit in the rules, as opposed to embedded in the interface code.

As an example, consider the problem of adapting the hotel-booking interface to a mobile environment. The designer has some choices to make: The first is to define a *different* Abstract Interface altogether for each device family; the second is to define a *generic* interface, and specialize it for each device family; and the third is a combination of both – define intermediary abstract interfaces for groups of families of devices based on common properties, and specialize one of them for each specific family. Our approach allows all three alternatives, allowing a better comparison among them, e.g., based on the complexity of the models used for each approach.

One frequent concern with rule-based architectures is performance. We are now in the process of systematically evaluating the performance overhead introduced by our approach. Nevertheless, within the Synth architecture[10], we have already observed that the overall application performance is not significantly affected by this interface architecture, because of the much larger performance hit caused by database access and inferencing while executing the business logic operations.

We are continuing this work in several directions. The first is to continue the evaluation of the approach, both in terms of performance, but also in terms of its expressivity and usability for developers. Second, we want to explore the design trade-offs for multi-platform applications, along the lines discussed in this section. Finally, we plan to extend the rule-based adaptation engine to encompass all models in SHDM besides the Interface Model, to achieve fully adaptive applications.

Acknowledgments. Daniel Schwabe was partially supported by CNPq (WebScience INCT).

[10] Synth currently uses the BigOWLIM RDF store.

References

1. de Souza Bomfim, M.H., Schwabe, D.: Design and Implementation of Linked Data Applications Using SHDM and Synth. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 121–136. Springer, Heidelberg (2011)
2. Calvary, G., et al.: The CAMELEON Reference Framework, CAMELEON Project (September 2002), http://giove.isti.cnr.it/projects/cameleon/pdf/CAMELEON%20D1.1RefFramework.pdf
3. Coutaz, J., Calvary, G.: HCI and Software Engineering for User Interface Plasticity. In: Jacko, J. (ed.) Human Computer Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, 3rd edn. Taylor and Francis Group Ltd. (May 2012)
4. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. In: Proc. of the WWW9 Conf., Amsterdam (May 2000)
5. Frasincar, F., Houben, G.J., Barna, P.: Hypermedia Presentation Generation in Hera, Information Systems, vol. 35(1), pp. 23–55. Elsevier Science Ltd., Oxford (2010)
6. Lima, F., Schwabe, D.: Application Modeling for the Semantic Web. In: Proceedings of LA-Web 2003, Santiago, Chile, pp. 93–102. IEEE Press (November 2003)
7. Helms, J., Schaefer, R., Luyten, K., Vermeulen, J., Abrams, M., Coyette, A., Vanderdonckt, J.: Human-Centered Engineering with the User Interface Markup Language, Human-Centered Software Engineering, ch. 7, pp. 141–173. Springer, London (2009)
8. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-based Web Engineering: An Approach based on Standards (book chapter). In: Rossi, G., Pastor, O., Schwabe, D., Olsina, L. (eds.) Web Engineering: Modelling and Implementing Web Applications, ch. 7, pp. 157–191. Springer, HCI (2008)
9. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: A Language Supporting Multi-path Development of User Interfaces. In: Feige, U., Roth, J. (eds.) EHCI-DSV-IS 2004. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005)
10. Meixner, G., Paternó, F., Vanderdonckt, J.: Past, Present, and Future of Model-Based User Interface Development. i-com 10(3), 2–11 (2011)
11. Myers, B., Rosson, M.B.: Survey on User Interface Programming. In: Proc. 10th Annual ACM CHI Conference on Human Factors in Computing Systems, pp. 195–202 (2000)
12. Nunes, D.A., Schwabe, D.: Rapid prototyping of web applications combining domain specific languages and model driven design. In: Proc. 6th International Conference on Web Engineering (ICWE 2006), pp. 153–160. ACM (2006) ISBN 1-59593-352-2
13. Paterno, F., Santoro, C., Spano, L.D.: Maria:A Universal, Declarative, Multiple Abstraction Level Language for Service-Oriented Applications in Ubiquitous Environment. ACM Transactions on Computer-Human Interaction (TOCHI) 16(4) (November 2009)
14. Silva de Moura, S., Schwabe, D.: Interface development for hypermedia applications in the semantic web. In: Proc. WebMedia and LA-Web, Ribeirão Preto, Brazil, pp. 106–113. IEEE Press (October 2004)
15. Virgilio, R., Torlone, R., Houben, G.J.: Rule-based Adaptation of Web Information Systems. In: Proc. 7th International Conference on Mobile Data Management (MDM 2006), Nara, Japan, May 10-12. Springer Science (2006)