Using Conditional Transformations for Semantic User Interface Adaptation

Pascal Bihler, Merlin Fotsing, Günter Kniesel
Institut für Informatik III
Universität Bonn
Römerstraße 164
53117 Bonn, Germany
{bihler, fotsing, gk}@cs.uni-bonn.de

Cédric Joffroy
Laboratoire I3S, CNRS
930, route des Colles BP 145
06903 Sophia Antipolis Cedex
France
joffroy@i3s.unice.fr

ABSTRACT
The rapid growth of mobile Internet use requires highly flexible and adaptable user interfaces for web applications. Contextual data from various sources as for example device HMI constraints, environmental information or device sensors need to be taken into account to dynamically adapt the application’s UI. The same is true for web applications that are based on context defined service orchestration. To tackle the complexity of multiple, possibly interacting adaptations of web user interfaces, the authors propose the use of semantic user interface descriptions [1] and automatic creation of adapted user interface code via logic-based Conditional Transformations [7, 10] of UI representations. With our approach, existing representations of context values in logic databases [16], can easily be included into the adaptation.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques—User interfaces; D.2.11 [Software Engineering]: Software Architectures—Data abstraction, Languages

Keywords
Abstract User Interfaces, LAIM, Conditional Transformations, Web Form Rendering

1. INTRODUCTION
Since mobile Web access is getting cheaper and the processing and visualization power of handsets has increased in the last years, Web-based applications constitute a serious alternative to classical mobile applications. The advantages for application developers are immanent: On the mobile device, only a standard-conform browser is required, which is often part of the device’s operating system or default application set. The provision of a URL is enough to enable the user’s access to the application, no special deployment process is required, as no special update procedure.1 Data sets can be stored on a powerful database server instead of being limited by the storage capacity of the mobile devices. Ideally, one instantiation of the program can be “executed” on all different mobile devices, eliminating the need of developing device specific binaries.

Unfortunately, not all devices can be handled the same way. Like it is true for desktop browsers, mobile web browsers interpret Web standards differently. Different devices have different interaction constraints (like screen size and resolution, input methods etc.). Additionally, mobile Web applications are used in a variety of user contexts – while a classical Web application “lives” in a fixed context, where the user is sitting in front of his PC, mobile applications are used in a variety of contexts: a user might decide to listen to streamed music while sitting in a café, jogging, waiting at the airport or even in the bathroom. For the same application, each usage scenario might impose a different UI view:

1Some devices even provide the option to save the URL in the application starter, supporting the perceived convergence of classical applications and Web applications.

To deal with different device capabilities, frameworks like the Wireless Abstraction Library (WALL) [14] allow an automatic adaptation of HTML code. In WALL, a configuration file controls the adaptation of tags in a page so that they are correctly interpreted by different device browsers. While this allows device specific HTML code adaptations, user context still needs to be managed by the application developer.

2. APPROACH OVERVIEW
To decouple the UI adaptation process from the application core and allow the sharing of UI adaptation functionality among several applications, we propose a solution based on semantic UI descriptions: Specifying the interface in an abstract way allows the Web application to delegate interface adaptations to a dedicated layer. This frees application programmers of dealing with the complexity of adaptations required by different device capabilities and contexts. Managing the entire cross-product of device and context specific adaptations, a true maintenance nightmare if per-
3. SEMANTIC UI DESCRIPTION

To handle different end-user preferences and different computing platforms we generate dynamically an adapted user interface based on a Semantic User Interface (SUI) [6] description. Herein we guarantee a separation in between core functionality, UI requirements and their visual appearance. Several languages for describing interfaces in an abstract or semantic way have been developed in the last years, e.g., XIML [15] and UsiXML [11].

Being a W3C standard candidate, UsiXML (User Interface eXtensible Markup Language) seems to be a natural choice for the semantic representation of user interfaces. The language is based on graph transformations and allows a representation of UI at different level of abstraction. Every artifact and transformation rule has a graphical syntax. Artifacts can be understood as formal functionalities to improve, which the physical runtime environment is taken into account. Concrete rules define operations producing relevant transformations on the graphical structure.

Unfortunately UsiXML is relatively complex. To handle its complexity would imply a considerable overhead even for small experiments. Therefore, we decided not to use UsiXML in a first approach, but to build upon a very simple semantic UI language we called LAIM.

LAIM (Language for Abstract User Interfaces) is simple but flexible. It consists of four main elements (Figure 3):

- **Output** Defines an output of any kind, which can be visually represented based on type/context information.
- **Input** An input is restricted by the data type it receives, optionally offering several default choices.
- **Action** An action allows to trigger program functionality and owns a caption (might be visualized as a button, menu or otherwise).
- **Group** With this container element, UI context can be described. This might help an engine to structure the rendered user interface and decide upon the proper visualization of the elements within the group.

Based on this simple, abstract description of UI functionality, we use CTs, as described in the next section, to create code that represents a concrete UI.
4. TRANSFORMATIONS

In this section we explain the motivation for adopting Conditional Transformations and explain how we use them.

4.1 Challenges

Because of the many variants of platform- and context-specific adaptations that may be necessary in arbitrary applications there is an exponential number of possible combinations. Therefore, implementing adaptations for specific combinations is prohibitive. Instead, we need the ability to compose arbitrary combinations from modular descriptions of individual adaptations and to infer automatically when and how such composition is possible. In particular, we must face the following challenges:

Modularity and compositionality It must be possible to specify each adaptation separately and to compose complex adaptations by reusing existing ones. There must be no need to modify existing adaptations for this purpose. In particular, compositionality must not be limited to a fixed set of anticipated combinations for which specific hooks have to be hand-coded into each adaptation.

Interaction detection and resolution If independently developed adaptations are deployed together, they might interact in ways not foreseen by their authors. To make composition possible, it is necessary to have an automated way of detecting and resolving interactions.

We found Conditional Transformations to be the only practical model transformation approach that meets these challenges (for a brief comparison to others see 4.4).

4.2 Conditional Transformations

Conditional Transformations (CTs) are a logic-based language and formalism for expressing arbitrary software transformations guarded by arbitrarily complex preconditions. The transformation part of a CT executed for all elements that fulfill the condition of the CT (for details see [7] and [10]). CTs provide a theoretical and practical basis for model transformations and model driven engineering. Compared to other model transformation approaches CTs provide a unique combination of features. Among others, they are

Purely Declarative Unlike imperative programs with side-effects (including Prolog programs that manipulate their own factbase) CTs have a well-defined, model-theoretic semantics. Hence CTs are easy to compose, analyse and optimize automatically.

Composable Different composition operators, of which some are unique to CTs, allow creation of complex programs from simple, reusable units. Composition is possible even if not anticipated by the designers of the existing CTs. No hooks need to be built into CTs to enable their reuse in unanticipated contexts.

Analysable Automated analysis and resolution of interactions between CTs is possible [9, 8] thus providing the basis for the related interaction analysis of adaptations expressed by CTs.

Multi-Directional A CT with N arguments is like N! functions, since each argument can be used as input or

```javascript
ct ( translateNodeGroup ( Id, Group, ListOfChildren ),
    condition ( [ laimGroup ( Id, Group, _ ) ],
                   transform_id ( Js, Id, NewId ) ),
    transformation ( [ add ( jsDiv ( NewId, NewGroup, ListOfChildren ) ) ] )
)
```

Figure 4: LAIM sample as fact representation.

output. For instance, the same CT can be used to transform an input element into a `jsText` element or to transform an output element into a `jsLabel` element. What happens only depends on the way the CT is called. Its definition does not have to be rewritten. CTs transform models of software artefacts represented, similar to Prolog, as logic facts [7, 10]. However, unlike in Prolog, deduction (model analysis) and model update do not interfere during the execution of CTs.

4.3 Using CTs

Since CTs transform logical facts, creation of adapted user interfaces starts by transforming our LAIM description to a logical representation. For example, an `Input` element is transformed into a `limInput` fact and its caption is transformed into a `limCaption` fact. Both are linked by the shared `Id mail_listing` in their first argument (Figure 4).

Once the logic-based model of our abstract user interface is available, we can apply CTs to transform it to a model of an output language program that can be translated to output language source code.

Figure 5 shows an exemplary CT to transform a `limGroup` element into a `jsDiv` element. In the condition, the given `Id` is verified as being an `Id of a limGroup element`. The involved `Ids` are transformed to assure uniqueness. If all conditions are passed, the `jsDiv` element of the output model is created.

At the current state of our research, we only used one-to-one transformations between LAIM elements and output elements. This demonstrates the applicability of the approach but is still far from a practically useful tool. For example, we currently do not translate LAIM inputs into lists or checkboxes, even if the semantics of the respective input might lend itself to such a representation. A LAIM input element will probably have a bunch of possible output representations. In future work, we will develop rules CTs for context-dependent transformations.
4.4 Comparison with other solutions

One approach to context-dependent user interface adaptation is Comet [3]. It adapts the UI in function of the usage context, defined by the triple

< User, Environment, Platform >

The adaptation is executed at runtime, taking into account the complete set of possible adaptations. This complexity might not be handled on a mobile device, especially if several applications are executed in parallel (which might lead to a computational explosion). Our strength is to perform the adaption on the server and send the adapted UI to the device.

The translation of a LAIM description to executable GUI code is basically a model transformation task. However, we do not know of any other model transformation approach that currently fulfills our requirements regarding compositionality and automated interaction analysis and resolution (see Section 4.1). In particular, XSLT, which is typically used for transformation of XML, provides none. Combinations of multiple transformations expressed in XSLT require global knowledge about the implementation of each participating XSL script. Since scripts can only interact via the transformed data or global variables, making different scripts work together requires changing their code (in any non-trivial examples). Similar comments apply to model transformation approaches such as ATL [5] and QVT [13], which rely mainly on an imperative execution model. None of the known approaches supports interaction detection.

5. SUMMARY AND OUTLOOK

With the use of Conditional Transformations we envision a decoupling of transformation rules and transformation engine. Using a logic based approach, transformations and the corresponding ruleset can easily be defined, and reused in different transformation contexts. The connection of our transformation framework with Prolog based context databases, like used in the Context Sensitive Intelligence (CSI) project [12], is straightforward. In addition, our work evaluates the application of CTs in runtime environments.

As this work in progress evolves, we focus our research on the following questions:

- How can we describe well known and proven user interface design standards in an intuitive logic way, so that they can easily be evaluated using CTs?
- How can we handle interaction with the UI on the back-end side? Event management and forwarding, combined with partially updating the User Interface constitute interesting challenges.
- How can the transformation be further optimized so that it fully comply with today's web standards?
- How to include the user feedback into the UI adaptation cycle, as introduced in [4]?
- Is it possible to use the CT-approach not only for output generation and adaptation, but also for UI composition, as we suggested for mobile applications in [2]?

6. REFERENCES