Automatic Dialog Mask Generation for Device-Independent Web Applications

Matthias Book, Volker Gruhn, Matthias Lehmann
Chair of Applied Telematics/e-Business, Dept. of Computer Science, University of Leipzig*
Klostergasse 3, 04109 Leipzig, Germany
{book, gruhn}@ebus.informatik.uni-leipzig.de, matlehmann@web.de

ABSTRACT

When building web applications for use on different devices, developers need to deal with a wide range of input/output capabilities that affect how users interact with the application: A dialog that can be completed in one step on a desktop client may have to be broken up into a number of steps on a small-screen mobile device. Since it is time-consuming to define all the possible dialog masks and dialog flow variants for different channels manually, it would be desirable to automate the adaptation of dialog masks and flows.

To address this need, we introduce the DiaDef language for the abstract, device-independent definition of the widgets in a dialog, and the DiaGen framework that automatically breaks this abstract dialog definition down into sufficiently small dialog masks for the users' mobile devices and incorporates them into suitable micro dialog flows that are generated at run-time in order to be handled by our Dialog Control Framework.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design tools and techniques—*user interfaces*; H.5.4 [**Information Interfaces and Presentation (e.g., HCI)**]: Hypertext/ Hypermedia—*navigation*

General Terms

Design, Human Factors, Languages

Keywords

Device Independence, Dialog Control

1. INTRODUCTION

The increasing capabilities of mobile devices and the widening coverage of cellular and wireless networks allow users to access web applications not only from their desktop PC, but also from PDAs, mobile phones and similar devices today. While their input/output capabilities are maturing in technical characteristics such as display resolution and color depth, or interface features such as auto-completion and character recognition, their display size remains (and will remain) small in order to remain portable.

The wide-ranging differences in display resolution provided by the mobile and stationary devices that can be used to access web applications impact the way users interact with those applications. Studies have shown that applications' usability suffers when the data presented or requested on a small screen is too complex [11, 17]. Consequently, complex forms that can be displayed on one page on a desktop browser should be broken up into several simpler pages on a mobile device. Users can then fill them in one after another in order to accomplish the same result, as illustrated in Fig. 1 using the example of a dialog that prompts the user for his address.

In [4], we already presented the Dialog Flow Notation (DFN) that can be used to specify such device-specific dialog flows, and the Dialog Control Framework (DCF) that decouples device-specific presentation layers from the device-independent application logic layer by controlling the web application's dialog flow according to the DFN specifications. To build a web application using this approach, developers had to implement device-independent action classes that contain the application logic, device-specific dialog masks (web pages) that contain the user interface, and a dialog flow specification indicating how the user interface and application logic interact.

Even though the DCF enables reuse of the application logic across all channels, developers still need to consider the capabilities of all devices that users may possibly use to access the application, and implement a number of different presentation channels for different screen sizes. This entails examining each large-screen dialog mask individually, decid-

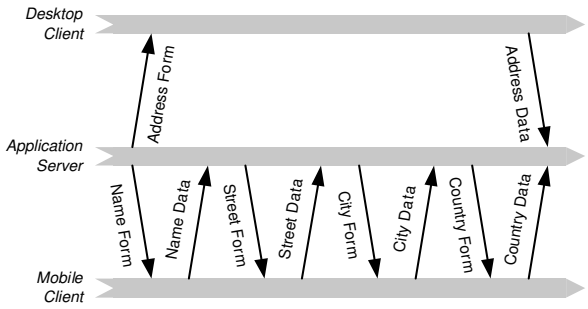

Figure 1: Dialog sequences on different devices.

*The Chair of Applied Telematics/e-Business at the University of Leipzig is endowed by Deutsche Telekom AG.

ICWE'06, July 11-14, 2006, Palo Alto, California, USA.
ACM 1-59593-352-2/06/0007.

ing if and where it may have to be broken up for small-screen devices, implementing the respective mask fragments, and specifying the dialog flow variants connecting those fragments. This manual implementation of the user interface for different devices (known as multiple-authoring with automated content selection [18]) consumes a lot of effort in the development process for multi-channel web applications.

In this paper, we therefore present a single-authoring approach with automated content adaptation in the form of dialog decomposition (pagination) and markup transformation. It comprises a language for the device-independent specification of dialog masks and a framework for the automatic generation of device-specific dialog masks and dialog flows at run-time. These extensions to the DFN and DCF allow developers to focus their efforts on implementing the application logic and specifying the user interface, without having to deal with their adaptation to a broad spectrum of devices. In the following sections, we will first present the dialog definition language DiaDef (Sect. 2) and then introduce the dialog generation framework DiaGen (Sect. 3), showing how it identifies the individual devices employed by users, breaks up the abstract dialog specification into suitable device-specific masks, creates the micro dialog graph connecting them, generates the concrete markup for a particular device and validates the incoming user input. Afterwards, we compare a plain DCF- and a DiaGen-based implementation of a small example dialog (Sect. 4) to evaluate the development effort and performance of the two approaches. We conclude with a comparison of DiaGen to related work in the field (Sect. 5) and an overview of further research opportunities (Sect. 6).

2. DIALOG DEFINITION LANGUAGE

Since different device capabilities require not only superficial layout changes, but also adapted dialog flows, as Fig. 1 shows, approaches that rely on transcoding the markup of an existing channel are not always sufficient. We therefore base our approach on an abstract dialog definition language that specifies only the functional and semantic properties of a dialog, but not its presentation [23]. At the time of generation of actual dialog masks, external parameters such as the device's display resolution and the application context determine how the dialog's abstract properties are presented, i.e. which widgets are used to implement certain functional properties, how widgets are laid out and paginated to convey semantic properties, and how constraints of the user's input are checked and enforced.

2.1 DiaDef Document Structure

While the abstract dialog definitions are specified by human developers, they must be processed by a framework and therefore be expressed in a format that is accessible to both. Thus, an XML-based language is the natural choice. After examining a number of languages (see Sect. 5), we chose XForms [9] as the basis of our DiaDef language since it provides language constructs for the realization of most of the above requirements already. DiaDef uses only a subset of the XForms standard (most notably ignoring all constructs dealing with client-server data exchange, such as most attributes of the `submission` element, since this is handled by the Dialog Control Framework), but also extends it with some specific constructs.

Each DiaDef document specifies the contents of one dialog in a web application, where we define a "dialog" as a collection of widgets that should all be displayed on the same hypertext page if a sufficiently large display is available (typically, one DiaDef document will correspond to one HTML page displayed on a desktop terminal, which may have to be broken up for smaller devices). The language does not allow the definition of hard, mandatory "page breaks" between widgets since this would introduce a dialog flow aspect into the specification – however, dialog flow specification is the sole responsibility of the Dialog Flow Notation [4], while DiaDef is solely concerned with the *contents* of dialog masks within that flow.

Since DiaDef documents can contain most language constructs of the XForms standard, we will only describe the DiaDef-specific extensions here. A DiaDef document always consists of two sections: A *data section* (in the `dd:data` element) that specifies the data model and functional properties of the data that the user is prompted for in this dialog, and an *interface section* (in the `dd:interface` element) that contains the abstract widget specifications.

The properties of the dialog's data model are expressed in terms of the `model` element and its children from the XForms Core module, i.e. an `instance` element containing model items, and `bind` elements associating properties such as constraints with those items. Besides the usual XForms Model Item properties, DiaDef also introduces the `dd:priority` attribute which can be used to assign a numerical priority value to any model item. The respective widget will then only be included in a generated page if the device requesting the page has been assigned a higher priority threshold in its profile (see Sect. 3.1). Developers could use this construct to assign optional fields a lesser priority, so they will not consume space on small screens.

The interface section contains the abstract specification of the widgets associated with the data model items, which are expressed using elements of the XForms Form Model module such as `input`, `select1` etc. The nestable `group` element of the XForms Group module can be used to specify semantic relatedness of widgets. If a dialog has to be broken up onto multiple pages, the DiaGen framework will attempt to keep grouped elements on the same page (the joint presentation can also be enforced by setting the `group` element's new `dd:dividable` attribute to `false`). Usually, the DiaGen framework will place the widgets on the hypertext page(s) in the order they are given in. However, in order to relax this restriction, developers may set the `group` element's new `dd:ordered` attribute to `false` to allow the framework to change the order of widgets, if that leads to

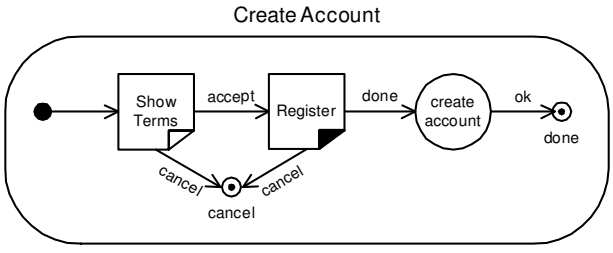

Figure 2: Macro dialog graph of *Create Account* module with abstract *Register* dialog, as specified by the developer.

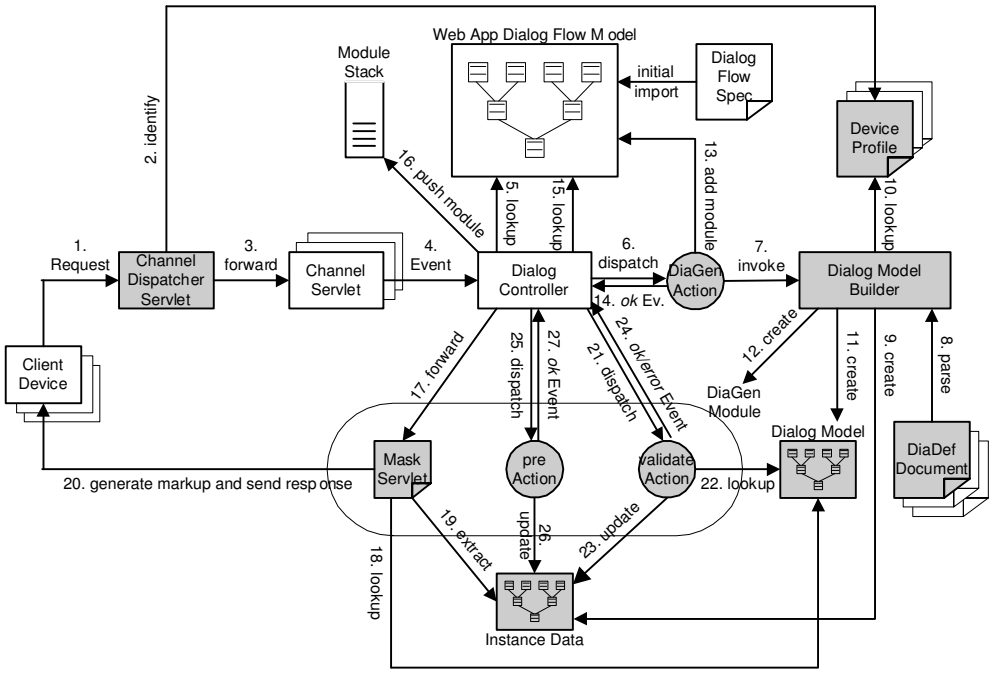

Figure 3: Architecture of the combined Dialog Control (white) and Dialog Generation (gray) frameworks.

a more space-efficient layout. Using the `dd:render-hint` element, developers can provide further information to the framework on how large certain fields should be rendered. In order to realize dynamic lists of widgets, the elements of the XForms Repeat module can be used as usual.

Since DiaDef documents are essentially XForms documents with a few extensions, we refrain from presenting a source code example here for the sake of brevity. We will however discuss the required implementation effort in Sect. 4.1.

2.2 Integration with the DFN

The integration of an abstract dialog specification such as the above with the DFN is quite straightforward: In the DFN's dialog graph diagrams, abstract dialogs are symbolized as sheets with a black dog ear (as opposed to concrete dialog masks, which are distinguished by a white dog ear) to convey their similarity to concrete masks: Abstract dialogs can be used in dialog graphs exactly the same way as concrete masks, as illustrated in Fig. 2. In this example of a *Create Account* module, *Show Terms* is a concrete mask that will be displayed on all devices as one hypertext page[1], while *Register* is an abstract dialog that may be broken up into several dialog masks on some devices. Independently of the number of masks that it may be broken into, the *Register* dialog will ultimately generate either a *done* or *cancel* event that determines if the account will actually be created.

When the DCF encounters an abstract dialog at run-time, the DiaGen framework will parse the respective DiaDef document and generate the necessary concrete masks and micro dialog graphs connecting them automatically, as described in the following section. We call the dialog graphs that are auto-generated by DiaGen to connect concrete dialog masks *micro dialog graphs* in order to distinguish them from the manually specified *macro dialog graphs* that the developer embedded the abstract dialog in.

[1]The original DFN and DCF already allow the specification of several device-dependent implementations for such concrete pages, however no multi-page implementations.

3. DIALOG GENERATION FRAMEWORK

The Dialog Generation Framework (DiaGen) is an extension of the Dialog Control Framework (DCF) [4]. It becomes active when an abstract dialog is encountered in a user's traversal of the web application's dialog flow. Fig. 3 shows the complete architecture of the combined frameworks, with the DiaGen components shaded gray. In the following sections, we will use this figure to describe several aspects of the framework in detail – the identification of the device requesting a mask (Sect. 3.1), the generation of a suitable micro dialog graph (Sect. 3.2) and its traversal, which includes generating concrete dialog masks and validating the user input according to the DiaDef specification (Sect. 3.3).

3.1 Device Identification

In order to handle requests coming in from various client devices suitably, we first need to classify the device that sent a request. The *channel dispatcher servlet* that receives each incoming request (step 1 in Fig. 3) uses the Delivery Context Library (DELI) [5] to retrieve information about the client's capabilities. For mobile devices, this can be achieved by evaluating User Agent Profile (UAProf) information [19] that is provided by the device in terms of the Composite Capability Preferences Profile (CC/PP) standard [15] and describes device properties such as display resolution, color depth, supported markup languages etc. If no UAProf information is provided by a client device, DELI can use the browser identification string transmitted in the HTTP header to try to match the client to a *device profile* already stored on the server (step 2). Using the property information, the device is then classified as belonging to one of the

presentation channels supported by the web application (e.g. rich HTML for desktop devices, light HTML for PDAs and WML for mobile phones).

The request is then forwarded to the appropriate *channel servlet* (3) that parses the parameters found in the HTTP request[2] and generates a device-independent dialog event carrying these parameters, which is sent to the *dialog controller* (4). The dialog controller looks the event up in the application's *dialog flow model* (5) that was initially built from the dialog flow specification and takes appropriate action, e.g. dispatching the event to an action, forwarding the request to a mask, nesting or terminating dialog modules, etc.

3.2 Micro Dialog Graph Generation

If the dialog flow model indicates that an event leads to an abstract dialog, the DCF dispatches the event to the DiaGen framework via the *DiaGen action* (step 6 in Fig. 3). This special action first invokes the *dialog model builder* (step 7), which is responsible for transforming the abstract dialog specification into concrete dialog elements that are suitable for the channel that the request came in on. For this purpose, the dialog model builder first parses the *DiaDef document* for this particular abstract dialog (8) and creates an *instance data* model (9), which will be used to store the information that the user enters in the concrete dialog masks temporarily while the whole dialog sequence is not yet completed.

Next, the dialog model builder looks up the display properties of the current channel in the device profile identified earlier by the channel dispatcher (10). This information is used to create the *dialog model* (11), an object hierarchy that contains instances representing all concrete dialog masks in the micro dialog graph, as well as the widgets placed on them. The dialog model is built by reading widget specifications from the DiaDef document, choosing appropriate widget implementations for the current channel and adding them to a page while estimating how much screen space they will occupy. DiaGen's current layouter implementations only make a quite coarse estimate based on the number of characters in a widget, which obviously still bears potential for optimization. When a page is "full" according to the device properties and the cumulative estimated widget size, a new page is added to the model and further widgets are added to that page, honoring the semantic rules regarding division, order and priority of widgets specified in the DiaDef document. Furthermore, validator instances for the various data types and constraints specified in the DiaDef document are added to the model. The implementation of this process in the DiaGen framework has been designed to be highly customizable: Developers can extend the provided layouter classes with their own layout algorithms and space estimation heuristics, which may be optimized for particular applications, and define application-specific data types, widgets and validators, while treating the rest of the DiaGen framework as a black box.

Once the description of the individual dialog masks in the dialog model is complete, the dialog model builder finally generates the micro dialog graph connecting these masks and their supporting actions for data storage and validation

[2]The DCF already provides HTML- and WML-processing channel servlets; others can be added by application developers as needed.

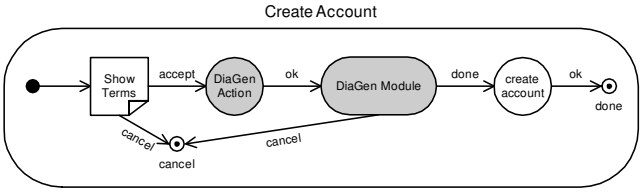

Figure 4: Macro dialog graph of *Create Account* module with abstract dialog resolved automatically into *DiaGen Action* and *DiaGen Module*.

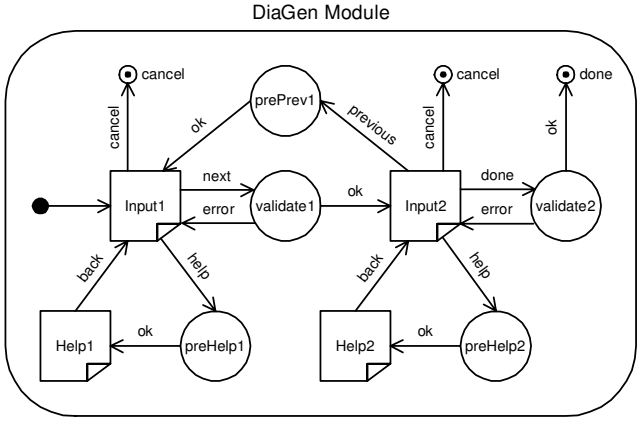

Figure 5: Auto-generated micro dialog graph of *DiaGen Module* with two concrete dialog masks.

(see Sect. 3.3) and encapsulates them in a *DiaGen module* (12). The DiaGen module is built out of DCF dialog element instances and can therefore be incorporated into the application's dialog flow model by the DiaGen action at the exact point where the abstract dialog was originally specified (13). When the DiaGen action terminates, it returns an *ok* event calling the DiaGen module to the DCF (14), which now "sees" the macro dialog graph in Fig. 4, consisting only of concrete dialog elements instead of the original abstract dialog.

3.3 Micro Dialog Graph Traversal

The DiaGen module contains the micro dialog graph that was generated from the abstract dialog specification for a particular channel. Figure 5, for example, shows the micro dialog graph for an abstract dialog that was broken into two concrete dialog masks. Those *Input* masks are connected with *next* and *previous* links according to the "wizard" interaction pattern [24]. Each mask also provides a link to a *Help* mask that may contain further information about the widgets on the referring mask, as well as a *cancel* link to terminate the wizard. The last mask in the wizard sequence provides one or more application-specific links instead of the *next* link (here, just the *done* link to complete the wizard and continue traversal of the macro dialog graph).

When the dialog controller looks up the receiver of the *ok* event from the DiaGen action (step 15 in Fig. 3), it now finds the newly generated DiaGen module in the dialog flow model and pushes it onto the stack of currently traversed modules (step 16). The module's initial event leads to the first *Input* mask. Normally, the dialog controller forwards the request to a JSP specified in the dialog flow model for

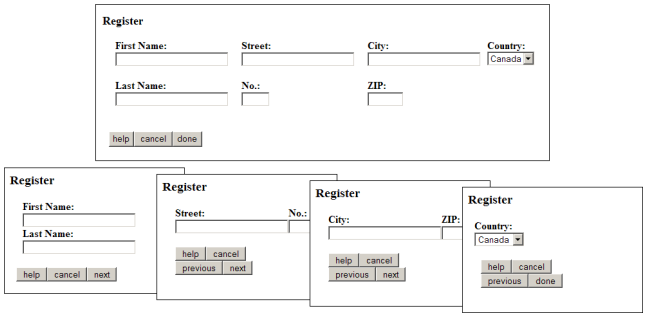

Figure 6: Concrete dialog masks for abstract *Register* dialog on PC (top) and PDA channel (bottom).

each mask, however, since this dialog graph was generated on-the-fly, no hard-coded mask implementation exists on the server. Instead, the dialog flow model contains a reference to the DiaGen framework's *mask servlet*. When the DCF forwards the request to this servlet along with the name of the mask to display (17), the servlet looks up the respective mask's contents in the dialog model (18), reads any preset values for the widgets from the instance data model (19) and then generates a hypertext page containing the widgets assigned to this mask, using the appropriate markup language for this channel. The dialog step concludes with the response being sent back to the client (20).

When the user submits the mask by clicking on the *next* button, the request is handled by the DCF as usual, which deals with the DiaGen module that is now on top of its module stack just like with any other module. The event is looked up in the dialog flow model and then dispatched to the *validate action* (21). This action looks up the data types and constraints defined in the dialog model for the preceding mask (22), and checks whether all rules are satisfied. If this is the case, the action stores the submitted data in the instance data model (23) and returns an *ok* event to the dialog controller (24). Otherwise, an *error* event is returned, which prompts the dialog controller to lead the user back to the same mask, where an error message is displayed.

In accordance with the "wizard" interaction pattern, if the user clicks on the *previous* or *help* button of any mask, the dialog controller dispatches the event to the *prePrev* or *preHelp* action according to the dialog flow model (25), which simply stores the data entered into that mask so far in the instance data model without validation (26) and then returns an *ok* event, which prompts the DCF to continue traversal of the micro dialog graph towards the previous *Input* or appropriate *Help* mask, as shown in Fig. 5.

When the user ultimately completes the dialog graph, the last *validate* action stores the contents of the instance data model in its outgoing event, which leads to the termination of the DiaGen module and the continuation of the macro dialog graph that it was embedded in (such as the one in Fig. 4). With its removal from the stack by the dialog controller, the DiaGen module is destroyed and will have to be rebuilt when the same abstract dialog is encountered again. This is necessary since the next request may be from a different user on a different channel or the same user in a different application context, which may influence the pagination and layout of the auto-generated concrete dialog.

It is important to note that the complex control flow that we just described is concealed by the DCF and DiaGen framework. Application developers only need to specify the device-independent abstract dialogs in DiaDef documents and implement any application-specific widgets and data types they wish to use, and the framework will take care of the actual device-specific pagination and micro dialog flow control. The dynamic generation of dialog masks and dialog graphs also remains transparent for users, who will only be exposed to the familiar "wizard" interface in situations where one dialog does not fit on a single page, as illustrated in Fig. 6 showing concrete masks generated from the abstract specification of the *Register* dialog.

4. EVALUATION

To evaluate the suitability of our approach for practical use, we need to examine three aspects: The effort required for developing dialogs, the performance of the framework, and the usability of the generated user interface. In the following subsections, we will address these aspects using the familiar *Register* example, which we implemented for a PC and a PDA channel, once using just the features of the original Dialog Control Framework [4] and once using the DiaGen extension presented in this paper. Both channels employ HTML, but while the PC channel assumes a display resolution of 1024x768 pixels, the PDA channel is restricted to 250x200 pixels. To make both versions comparable, we manually re-implemented the same layout and micro dialog flows in the DCF-based version that the DiaGen-based version had auto-generated.

4.1 Development Effort

In order to build the *Register* dialog manually in the DCF-based version, we had to implement one input JSP for the PC channel and four input JSPs for the PDA channel, as well as seven Java classes for validating and saving the user input individually. Additionally, we had to spell out the dialog flow connecting these masks and actions in the Dialog Flow Specification Language (DFSL) documents representing our DFN model [4]. In total, we had to write 573 lines of code (LOC) or 20,114 bytes in 15 files. In contrast, for the equivalent DiaGen-based implementation, we just had to specify the characteristics of the *Register* dialog in a DiaDef document and reference this specification in the application's DFSL documents, which required a total 95 LOC or 2,775 bytes in 3 files. No implementation of masks or actions was necessary.

Comparing the total size of both implementations, we see that the DiaGen implementation of the *Register* dialog is about 85% smaller in terms of LOC or file size. Although these are not the only factors determining the overall development effort of an application, they do give an indication of the amount of specification and coding work to be done. Also, while the absolute LOC and file sizes measured in this small example are obviously not representative for large-scale applications, we hypothesize that the relation between the DCF-based and DiaGen-based implementations can be extrapolated to larger applications, which are composites of similar dialog fragments and dialogs as those in our example.

4.2 Application Performance

The DiaGen framework facilitates the implementation savings discussed above by auto-generating the dialog flow, dialog masks and validation code that would otherwise have

Table 1: Average performance of DCF-based vs. DiaGen-based dialog flow implementation.

	DCF-based		DiaGen-based		Difference	Factor
Mask	Device Ident	Dialog Control	Device Ident	Dialog Control	Dialog Control	Dialog Control
PC Channel						
Terms Mask	0.0 ms	4.6 ms	713.2 ms	5.4 ms	0.8 ms	1.2
Register Mask	0.0 ms	5.4 ms	709.1 ms	61.0 ms	55.6 ms	11.3
Summary Mask	0.0 ms	5.6 ms	675.3 ms	7.0 ms	1.4 ms	1.3
PDA Channel						
Terms Mask	0.0 ms	4.5 ms	642.4 ms	4.8 ms	0.3 ms	1.1
Register Mask (1)	0.0 ms	4.3 ms	644.5 ms	42.0 ms	37.7 ms	9.8
Register Mask (2)	0.0 ms	4.6 ms	657.2 ms	19.7 ms	15.1 ms	4.3
Register Mask (3)	0.0 ms	4.6 ms	767.9 ms	16.4 ms	11.8 ms	3.6
Register Mask (4)	0.0 ms	4.7 ms	652.6 ms	17.2 ms	12.5 ms	3.7
Summary Mask	0.0 ms	5.9 ms	643.1 ms	6.3 ms	0.4 ms	1.1

to be hand-coded. As described in Sect. 3.2, much of this auto-generation happens at run-time when requests come in, since the dialog masks are tailored to the characteristics of the individual devices employed by users. Since the logic required to identify the device, generate the micro dialog flow and populate its masks with widgets is quite complex, the cost of these computations must be considered.

To evaluate the performance implications of the presented approach, we compared the response time of the DCF-based implementation of our example dialog flow to the DiaGen-based implementation, using a dialog sequence comprising the *Show Terms* mask, the actual *Register* dialog (with one mask on the PC channel, and distributed across four masks on the PDA channel), and a summary mask showing all entered information. This dialog sequence was traversed 17 times for each implementation (DCF- and DiaGen-based) on each channel (PC and PDA), with the server being restarted prior to each of those four test runs. The timings of the first two dialog sequence traversals in each test run were discarded since they were massively distorted by the compilation of the JSP pages and the initialization of the frameworks, while the timings of the 15 remaining traversals in each test run were averaged.[3]

Table 1 shows the averaged timings for requesting each mask in the PC and PDA dialog sequence through the DCF-based and the DiaGen-based implementation. The total time for each request is composed of the time required to identify the device (given in the *Device Ident* columns) and the time required to generate the mask and control the micro dialog flow (given in the *Dialog Control* columns). The first *Device Ident* column contains all zeroes since the original DCF does not perform any device identification by itself. Rather, it relies on developers to hard-code the references to the desired channel servlets into their mask implementations. Since all masks were implemented as individual JSPs in the DCF-based approach, they all took about the same time to request (approximately 5 ms on our test platform).

For the DiaGen-based implementation of our example, we observed much longer total response times of about 700 ms. As the table shows, most of this time was incurred by the device identification logic. Since this is largely contained in the DELI framework [5], it is beyond the direct influence of DiaGen. However, we are confident that the device identification time can still be optimized through suitable configuration and adaptation of DELI.

Focusing on the actual DiaGen logic, we observe that it increases the time required for dialog control notably, yet not extremely: On both channels, building the first mask of the *Register* dialog took roughly ten times longer than if it was implemented directly as a JSP; and the dialog generation and control logic for the subsequent masks on the PDA channel then took about four times as long as the delivery of the corresponding hand-coded JSPs. This is due to the fact that when the *Register* dialog was first requested, the DiaGen framework had to figure out a suitable pagination and generate the respective micro dialog flow, while on subsequent requests, the mask servlet responsible for generating the concrete dialogs merely had to assemble the markup according to the layout determined in the first step. Finally, note that dispatching the request to the non-DiaGen-generated *Terms* and *Summary* masks took about the same time as in the DCF-based implementation (once the device had been identified by DELI).

Summing up the results, we can say that using DiaGen instead of manually implementing the *Register* dialog introduced a ten-fold increase in the dialog control time for the first mask, and a four-fold increase for the subsequent masks in that dialog. While these may sound like huge performance penalties at first, the absolute increase in dialog control time observed in this example was only about 56 ms on the PC channel, as well as about 38 and then 15 ms on the PDA channel, which is still acceptable since the time lag introduced by the network is often higher. Of course, the actual timings observed in a production environment will depend heavily on the complexity of the dialog specifications, the complexity of the layout algorithms, and the execution speed of the underlying platform. In our ongoing research, we are striving to gain performance data for DiaGen-based systems running under production conditions.

[3]All timings were taken with the frameworks deployed on Sun Java Application Server 8.1 installed under Microsoft Windows XP Professional SP2 on a PC equipped with 1 GB of RAM and an Intel Pentium III processor running at 1.2 GHz. To achieve accurate measurements, we employed a native timing library that relies on the CPU clock cycle counter and CPU frequency to provide sub-millisecond precision. When interpreting the results, it should be noted that the operating system's time slicing is bound to introduce some jitter into the timings, and a production-quality server platform would likely yield lower values. However, for the purpose of this evaluation, we were less interested in the absolute measurements and more in the relationship between the timings of the DCF- and the DiaGen-based implementation.

4.3 Interface Usability

The last aspect of interest in the evaluation of the DiaGen approach is the usability of the auto-generated dialogs, which is mainly characterized by the dialog masks' layout and the structure of the micro dialog flow connecting them.

Since our current version of the DiaGen framework implements only rudimentary layout algorithms, the generated dialog masks do not yet come close to the usability that can be achieved by hand-coding JavaServer Pages. However, DiaGen was designed so that new layout algorithms (encapsulated in their own `Layouter` classes) can be added easily by registering them in a configuration file. The development of more sophisticated layout algorithms is a topic of our ongoing research, which will also include a formal usability test of manually implemented vs. DiaGen-generated masks as soon as those algorithms are sufficiently mature.

We do not foresee any usability problems stemming from the structure of the underlying micro dialog flow connecting the auto-generated masks, since it mirrors the "wizard" interaction pattern [24] with back and forward navigation, help and cancel options that users have long becomed accustomed to from window-based applications. This hypothesis, however, will still have to be examined in formal usability tests using real-life applications.

5. RELATED WORK

A number of approaches currently strive to provide device-independent access to web applications. Often, however, they focus on adapting static content for different devices. For example, the transcoding proxies Digestor [3], Wingman [10] and PowerBrowser [6] transform existing web content for display on mobile devices without having to change the original documents. However, these approaches are not suitable for interactive applications since any modification of their dialog masks (most notably in the form of pagination) requires an according modification of the underlying dialog control logic, which is not possible with a proxy-based approach.

In contrast to these approaches that provide "a posteriori" device independence, the Device Independence Web Application Framework (DIWAF) [13] enables device-independent authoring from the start. However, while it can deal with basic input forms, it is more geared towards static web content than to highly interactive web applications. SUPPLE [12] also generates user interfaces automatically based on device properties and user behavior. However, since SUPPLE was designed primarily for device control, important concepts for web applications such as dialog control and input validation are not supported. A strict separation of user interface and application logic is the goal of the Sisl approach [2], which takes care of collating the more or less finely grained input coming from a wide spectrum of input devices that use the same application logic. However, Sisl is not concerned with the pagination of dialogs for those devices.

The Renderer-Independent Markup Language (RIML) [25] is a language profile that extends XHTML and XForms with semantic information about dialogs that is processed by a pagination engine. While RIML provides more language constructs for the specification of dialog contents (e.g. regarding layout and navigation) than DiaDef does at this time, the combined DiaGen and DCF frameworks provide more dialog control features (e.g. nested dialog modules, abort and resume mechanisms) than current RIML implementations.

The Hypermedia Model Based on Statecharts (HMBS) [8] allows the channel-dependent specification of navigation patterns, however, it focuses especially on challenges such as synchronization that are introduced by multimedia elements embedded into hypertext. Schewe et al. [21] use a formal approach for modeling interaction and media objects that allows the specification of device-specific variants of media objects depending on the presentation channels' capabilities, but do not provide means for the automatic pagination of dialogs or generation of micro dialog flows.

Similar restrictions apply to other modeling approaches such OOHDM [22], UWE [16] and WebML. The latter, for example, is capable of modeling the layout and appearance of web pages independently of the output device, and allows developers to specify site views with different hypertext structures for different devices, but still requires developers to model those variants explicitly instead of generating them automatically [7].

As a basis for DiaDef, we considered a number of candidates: XHTML [1] is striving to move the Hypertext Markup Language further away from document design and more towards document semantics, however, it is not suitable as an abstract dialog description language since it still focuses too much on the presentation and too little on the functionality of forms. The User Interface Markup Language (UIML) [14] is a meta language that needs to be extended by a suitable vocabulary. However, since we would need an abstract vocabulary for our abstract dialog definition, the additional meta level would introduce unnecessary added complexity. Also, UIML does not provide an explicit data model that different user interface representations can share. Finally, the Extensible Interface Markup Language (XIML) [20] describes user interfaces in terms of relations between task, domain, user, presentation and dialog elements. However, specifying extensible data types, constraints, and dynamic widgets are more complicated to express in XIML than in XForms, which we ultimately based DiaDef on.

We did not examine other languages such as AAIML, AUIML, XAML and XUL in detail due to limited availability of documentation or a focus on a different user interface paradigm (e.g. window-based applications).

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the DiaGen framework, an extension to our Dialog Control Framework that allows web application developers to specify device-independent dialogs in the DiaDef language, and then breaks these abstract dialogs into one or more concrete dialog masks at run-time depending on the device that the user is working with. The dialog masks are incorporated into so-called micro dialog graphs that allow wizard-style navigation between the masks, ensure the validation of user input and provide context-sensitive help pages. This way, developers do not have to spend effort on the individual pagination and implementation of dialog masks for different devices, but can focus on the implementation of the application logic and data model.

Our implementation of the DiaGen framework shows that the core approach is feasible and extensible, and can be integrated smoothly with a state machine-based dialog control logic such as the one provided by the DCF. The initial evaluation of a small-scale example indicated that the approach

presented here can decrease the volume of user interface-related code and specifications considerably, while incurring an acceptable increase in the execution time of the dialog control logic.

The usability of the auto-generated dialog masks depends heavily on the availability of suitable profiles for associating devices with presentation channels, the accuracy of the heuristics used to estimate the size of different widgets, and the layout algorithms employed to render masks in different markup languages. These areas still bear a lot of potential for optimization and evaluation, which is a prime topic of our ongoing research. Possible improvements include extensions of the DiaDef language that allow developers to specify more semantic information to support the layout algorithms; and greater control over the generated micro dialog flow to enable other interaction patterns than just wizard-style navigation.

In parallel to our work on optimizing the pagination and layout algorithms, we are striving to gain experience with the use of the DiaGen framework in practice, both from a developer and a user point of view. Such case studies will enable us to evaluate the actual development effort and interface usability of DiaGen-based web applications in a more realistic setting.

7. REFERENCES

[1] M. Baker, M. Ishikawa, S. Matsui, P. Stark, T. Wugofski, and T. Yamakami. XHTML Basic W3C Recommendation. http://www.w3.org/TR/xhtml-basic/, 2000.

[2] T. Ball, C. Colby, and P. Danielsen. Sisl: Several interfaces, single logic. *Intl Journal of Speech Technology*, 3(2):91–106, 2000.

[3] T. W. Bickmore and B. N. Schilit. Digestor: Device-independent access to the world wide web. *Computer Networks and ISDN Systems*, 29(8–13):1075–1082, 1997.

[4] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *19th IEEE Intl Conf on Automated Software Engineering (ASE 2004)*, pages 100–109. IEEE Computer Society Press, 2004.

[5] M. H. Butler. DELI: A DElivery context LIbrary for CC/PP and UAProf. http://www.hpl.hp.com/techreports/2001/HPL-2001-260.html, 2001.

[6] O. Buyukkokten, H. G. Molina, A. Paepcke, and T. Winograd. Power browser: Efficient web browsing for PDAs. In *Proc Conf on Human Factors in Computing Systems (CHI '00)*, pages 430–437. ACM Press, 2000.

[7] S. Ceri, F. Daniel, and M. Matera. Extending WebML for modeling multi-channel context-aware web applications. In *Proc WISE – MMIS'03 Workshop (Mobile Multi-channel Information Systems)*. IEEE Press, 2003.

[8] M. C. F. de Oliveira, M. A. S. Turine, and P. C. Masiero. A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems*, 19(1):28–52, 2001.

[9] M. Dubinko, L. L. Klotz Jr., R. Merrick, and T. V. Raman. XForms 1.0 W3C Recommendation. http://www.w3.org/TR/xforms/, 2003.

[10] A. Fox, I. Goldberg, S. D. Gribble, and D. C. Lee. Experience with Top Gun Wingman: A proxy-based graphical web browser for the 3Com PalmPilot. In *Proc IFIP Intl Conf on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 5–18. Springer Verlag, 1998.

[11] G. Buchanan, M. Farrant et al. Improving mobile internet usability. In *Proc 10th Intl World Wide Web Conference*, pages 673–680. ACM Press, 2001.

[12] K. Gajos and D. S. Weld. SUPPLE: automatically generating user interfaces. In *Proc 9th Intl Conf on Intelligent User Interfaces*, pages 93–100. ACM Press, 2004.

[13] F. Giannetti. Device Independence Web Application Framework (DIWAF). http://www.hpl.hp.com/techreports/2002/HPL-2002-264.html, 2002.

[14] Harmonia, Inc. UIML v3.0 Draft Specification. http://www.uiml.org/specs/uiml3/DraftSpec.htm, 2002.

[15] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler, and L. Tran. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0 W3C Recommendation. http://www.w3.org/TR/CCPP-struct-vocab/, 2004.

[16] N. Koch and A. Kraus. Towards a common metamodel for the design of web applications. In *Proc 3rd Intl Conf on Web Engineering (ICWE 2003), LNCS 2722*. Springer, 2003.

[17] M. Jones et al. Improving web interaction on small displays. *Computer Networks*, 33:1129–1137, 1999.

[18] S. H. Maes. A 'single authoring' programming model: The interaction logic. In *Proc 2002 Symp on Applications and the Internet (SAINT)*, pages 12–13. IEEE Computer Society Press, 2002.

[19] Open Mobile Alliance. OMA User Agent Profile. http://www.openmobilealliance.org/release_program/uap_v20.html, 2003.

[20] A. Puerta and J. Eisenstein. XIML: A Universal Language for User Interfaces. http://www.ximl.org/documents/XimlWhitePaper.pdf, 2001.

[21] K.-D. Schewe and B. Thalheim. Modeling interaction and media objects. *Proc 5th Intl Conf on Applications of Natural Language to Information Systems, LNCS*, 1959:313–324, 2001.

[22] D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Comm ACM*, 38(8):45–46, Aug 1995.

[23] S. Trewin, G. Zimmermann, and G. Vanderheiden. Abstract user interface representations: How well do they support universal access? In *Proc 2003 Conf on Universal Usability*, pages 77–84. ACM Press, 2003.

[24] M. van Welie and H. Traetteberg. Interaction patterns in user interfaces. In *7th Pattern Languages of Programs Conference (PLoP 2000)*. Washington University, 2000.

[25] T. Ziegert, M. Lauff, and L. Heuser. Device independent web applications — the author once – display everywhere approach. In *Proc 4th Intl Conf on Web Engineering (ICWE 2004), LNCS 3140*, pages 244–255. Springer Verlag, 2004.