Mobile Business Processes

Volker Gruhn, Matthias Book

Chair of Applied Telematics/e-Business[1], Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany
{volker.gruhn, matthias.book}@informatik.uni-leipzig.de

Abstract. Today's global markets demand global processes. Increasingly, these processes are not only distributed, but also contain mobile aspects. We discuss two challenges brought about by these mobile business processes: Firstly, the need to specify the distribution of processes across several sites, and secondly, the need to specify the dialog flows of the applications implementing those processes on mobile devices. To remedy the first challenge, we give an overview of the Process Landscaping method with its support for refining processes across multiple abstraction layers and associating their activities and objects with distinguished locations. Next, we present a Dialog Flow Notation and Dialog Control Framework for the specification and management of complex hypertext-based dialog flows. These tools allow developers to build user interfaces for mobile client devices with different input/output capabilities, which all access the same application logic on a central server.

1 Introduction

The market reach of goods and services is ever increasing today – both in the business-to-consumer (B2C) and business-to-business (B2B) sector, transactions are performed on a regional, national or even global scope [22]. The global markets demand global business processes in order to handle those transactions efficiently. However, when looking at global markets, it would be a costly over-abstraction to consider the associated business processes as centralized entities [19]. Rather, they involve distributed teams, distributed service provisioning, and distributed repositories. This environment places higher demands on the infrastructure, coordination, communication and cooperation of the involved parties, all of which affect the suitable process models substantially ([10], [20]). As illustrated in the examples of the Iridium software process and housing industry processes, distribution affects both processes and data.

Recently, an additional challenge has been developing: As working environments are becoming more mobile, we are not just dealing with *distributed* processes anymore. In addition, we need to consider *mobile* processes: All sales-oriented processes tend to become more mobile, and the same is true for processes spreading over various sales channels. Also, processes delivering services to customers' locations tend to encompass mobile aspects. These mobile processes require flexible support for coordination and communication among the involved parties, as well as controlled remote

[1] The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

data access. Under these conditions, a central question is whether the mobility requirements affect the process models themselves or just their execution support [12].

1.1 Mobile Process Landscaping

When modeling the processes of a project, there are some key issues that developers need to resolve: After identifying the core processes, they need to determine a suitable order for modeling them and establish the interfaces between them. With regard to distributed and mobile processes, two especially vital questions are where process parts or activities are to be executed, and which data are needed in which location.

To support the specification, optimization and implementation of distributed processes, the Process Landscaping methodology was developed [13]. It comprises a number of activities that are also suitable for handling mobile business processes. The first step consists of identifying the high-level process clusters, positioning them in a process landscape and establishing their mutual interfaces (Fig. 1).

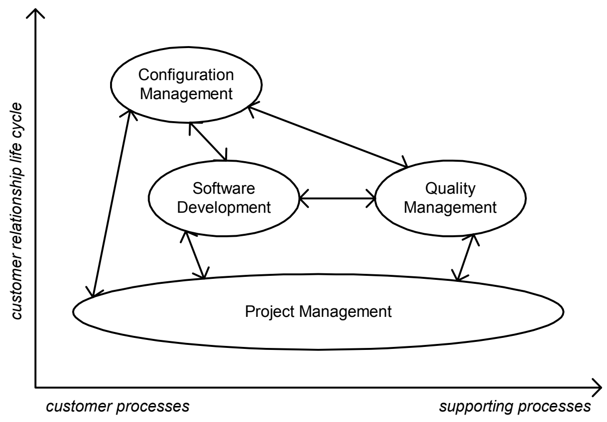

Fig. 1. Process landscape of a software development project (example)

In the following steps, different aspects of the process model are refined in whichever order is most natural in a concrete project: To refine interfaces, the data exchanged between the clusters is specified in combination with the direction of the data flow. Clusters can be refined in two different ways: The developer can either specify a set of sub-clusters that make up a super-cluster, or define a concrete process model that defines the activities performed and deliverables produced in a cluster. Activities in the process model may again be refined by sub-process models. This way, developers can move from a very coarse to a highly detailed definition of processes in a structured way – the overall process landscape serves as an orientation, with refinements being added on lower levels of abstraction as needed. This facilitates easy analysis of the model and discussion of the process.

Besides the structure imposed on the process model by the relationships between super- and sub-clusters, super- and sub-process models, further structural information is specified by assigning locations on the process landscape to objects and activities: Each activity must be assigned to one or more execution locations, and every object type must be assigned to a storage site. Furthermore, the interfaces are first-class entities in Process Landscaping to allow early identification of process relationships [14].

1.2 Mobile Process Implementation Characteristics

Despite the support by modeling methodologies such as Process Landscaping, the implementation of mobile processes is still hindered by a number of major obstacles today: Firstly, the required telecommunications infrastructure might be unavailable or unable to provide the necessary bandwidth. With network availability being less of a problem today (except for some isolated areas) and the introduction of high-volume transmission technologies such as UMTS imminent, this obstacle is starting to fade – however, slow deployment of the network equipment and mobile devices, combined with potentially high introductory prices, will likely limit the speed of adoption of mobile applications for some time to come.

Secondly, the currently employed legacy systems may be too inflexible for immediate integration with mobile processes, and difficult to open up to new access patterns. While not impossible, building suitable interfaces to integrate legacy systems with mobile processes and application front-ends is likely to be a complex and costly task. Similarly, organizational issues and traditional processes may not be compatible with mobile business processes and need to be adapted carefully to realize the full potential of mobile applications.

Finally, among the variety of mobile devices available today, only few mainstream conventions or *de facto* standards have developed yet. Since devices differ widely in aspects such as screen size, input/output interfaces, networking, programming and dialog capabilities, mobile applications either have to cater to the lowest common denominator, or be modified to fit different mobile devices. This becomes most obvious (and challenging) in the area of mobile dialog design.

One approach to solving these problems seems to be the use of hypertext-based user interfaces (UIs) for mobile applications, where the UI consists of web pages presented in a browser. Compared to window-based user interfaces, they require only modest client capabilities, making them especially suitable for mobile devices with their strict energy, memory, input and output limitations [9]. Furthermore, the simple information elements and interaction techniques of hypertext-based UIs can be rendered on various presentation channels, ranging from desktop to mobile devices [3]. This multi-channel thin client scenario requires the application logic to be implemented presentation channel-independently on a central server, while the UI is rendered individually on various client devices [23].

However, when developing applications with hypertext-based UIs, software engineers need to be aware that their implementation differs in some important characteristics from applications with window-based UIs ([21], [26]):

Firstly, the devices' different input and output capabilities restrict the amount of information users can work with at a time. Consequently, presentation channel-independent applications must not only implement different UIs, but also be able to handle different interaction patterns – for example, a task that may be completed in one interaction step with a desktop browser may take three steps on a mobile device and a dozen over an interactive voice response (IVR) system (Fig. 2).

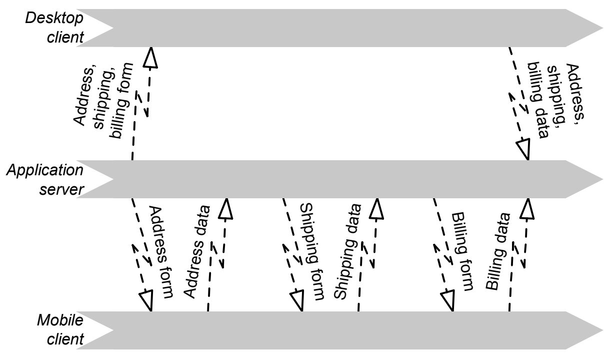

Fig. 2. Different dialog flows on different devices

Secondly, hypertext-based UIs present information on pages instead of in windows. Consequently, interactions that would be performed without involving the application logic in a window-based UI (e.g. closing a window) require the generation of a new page in a page-based UI and thus involve the application logic for every interaction step. Thirdly, hypertext-based UIs employ a request-response mechanism to pull data from the server. Since the application logic cannot push data to the client, it can only react passively to user actions (like clicking on a link) instead of actively initiating dialog steps (like opening a new window). Finally, the Hypertext Transfer Protocol (HTTP) is stateless: The protocol only transports data, but does not maintain any information on the state of the dialog system. Consequently, the application itself has to manage the dialog state for each user session, which requires complicated logic for more complex dialog structures.

Regarding the impact of these characteristics on the user experience, one of the most notable effects is the limitation to simple dialog structures in many hypertext-based applications today: Linear and branched dialog sequences can be easily implemented and are therefore commonplace, but already simple nested structures (e.g. an authorization form inserted at the beginning of a sensitive transaction) require a lot of dialog control logic, and no application that the authors are aware of is capable of nesting arbitrary dialogs on multiple levels.

Since users have a long-established conceptual model of nested dialogs from window-based applications, they will likely transfer that model to hypertext-based applications. However, because of insufficient dialog control logic, many applications still violate users' expectations today when they send them to other pages than they intended to reach (in some web applications, for example, login forms return users to the homepage after a successful login instead of sending them to the area that required authorization, forcing them to navigate manually to the desired area). This violation of the ISO dialog principles of controllability and conformity with user expectations [17] imposes a high cognitive and memory load on the user.

Since these challenges are independent of a specific application, a desirable solution would be a notation and a framework that can be used for the specification and implementation of any hypertext-based application. After giving an overview of the related work (section 2), we will therefore introduce a Dialog Flow Notation for specifying complex dialog flows (section 3), and present the architecture of a Dialog Control Framework for managing those dialog flows on different devices (section 4).

2 Related Work

A number of notations for the specification of interactive systems' user interfaces have been proposed over time. However, many were developed for traditional window-based applications and are therefore not suitable for the task of modeling the special characteristics of hypertext-based applications presented in section 1.2: While they can model direct manipulation techniques and multiple windows, which hypertext applications lack, they do not provide means for specifying request-response interaction patterns on page-based media.

Other approaches that were explicitly designed to describe hypertext systems mostly focus on *data-intensive* information systems, but not *interaction-intensive* applications [8]: For example, the RMM development process [16] allows the definition of navigable relationships between data entities, and the OOHDM [24] process provides classes like node, link and index to represent different forms of navigation; however, the resulting structures remain "flat" and cannot be nested arbitrarily. The same is true for the HDM-lite notation used by the Autoweb tool [7], which supports the automatic generation of database schemas and application pages from a conceptual model; and the modeling language DoDL [6], which allows mapping of structured database content to static hypertext pages, but does not support dynamic features. Finally, while the language WebML [5] is capable of modeling simple dynamic features of a data-intensive web application by providing operation units for creating, deleting and modifying entities, it does not support more complex structures such as modular, nestable dialog sequences.

For the implementation of hypertext-based applications, several frameworks exist that separate the user interface from the application logic to facilitate easier dialog control, as suggested by the Model-View-Controller (MVC) design pattern [18]. The Apache Jakarta Project's Struts framework [1] is one of the most popular solutions today. However, Struts forces developers to combine dialog control logic and application logic in the Model implementation, since the Controller does not implement any actual dialog control logic, but merely maps action names to class names (a more thorough discussion of the Struts approach vs. the one suggested in this paper will be presented in section 4).

The challenges of device-independent design are addressed in the Sisl (Several Interfaces, Single Logic) approach [2]. It inserts a so-called "service monitor" between the central application logic and the presentation logic for each device type to coordinate the events that the interface can generate with the events that the application logic can currently handle. This allows Sisl to support a wide spectrum of devices, including speech recognition systems, and handle the partial or unordered input that they may produce. However, since Sisl uses acyclic graphs for modeling dialogs, it seems more suitable for simple prompt- or menu-based interaction scenarios than for highly interactive applications with complex (i.e. nested or cyclic) dialog structures.

We are still missing a solution that controls the dialog structure of a hypertext-based application independently of the implementation of the Model and View tiers, supports different interaction patterns on different devices, and allows developers to work with complex dialog constructs like dialog modules nested on multiple levels. The Dialog Flow Notation and Dialog Control Framework introduced in the following sections are designed to address this need.

3 Dialog Flow Notation

To define the concept of a "dialog flow" and develop the elements of the Dialog Flow Notation (DFN), we first examine the client-server communication taking place when users work with a hypertext-based application. As Fig. 3 shows, a page *A'* displayed on the client is rendered from source code (e.g. HTML) that was first generated by an entity *A* (e.g. a JavaServer Page) on the server and then transmitted to the client. When the user follows a link or submits a form on this page, the resulting data *a* is transmitted to the server. The application logic may now process the data in a number of steps (here: *1* and *2*), which each generate data (*b* and *c*) that is processed in the next step. Finally, the source code for the following page is generated (*B*), transmitted to the client and rendered there (*B'*). Alternatively, user-submitted data (such as *d*) may not require any application logic processing, but directly lead to the generation and rendering of a new page (*C* and *C'*).

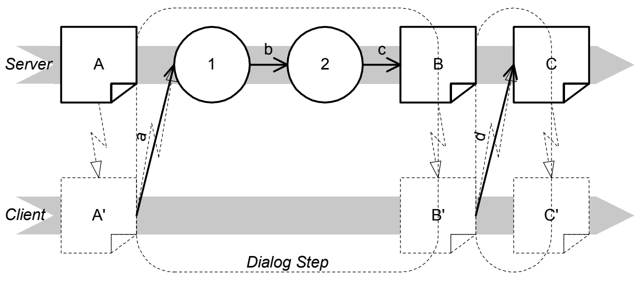

Fig. 3. Client-server communication in HTTP

We call the server activity happening between the submission of a request and the receipt of a response by the client a *dialog step* (in an online shop, for example, a dialog step might begin with submission of the user's billing information, comprise the validation of his credit card data by the application logic, and end with the generation of a confirmation page). Multiple consecutive dialog steps form a *dialog sequence* – for example, an online shop's checkout dialog sequence might be composed of several dialog steps for collecting the user's address, shipping options, and billing information. Finally, all possible dialog sequences that can be performed on a certain presentation channel of an application constitute that channel's *dialog flow*. An online shop's dialog flow might for example comprise searching for products, looking at detailed product information, putting products into the cart, checking out, etc.

3.1 Notation Elements

Looking back at the communication model in Fig. 3, we realize that the client-server communication and thus the distinction between generating (*A*) and rendering pages (*A'*) is irrelevant for the purpose of modeling dialog flows: When specifying how the user interacts with the application logic via the UI pages, the dialog flow designer does not need to know about technical details such as pages' source code being generated on the server and transmitted to the client prior to rendering.

The DFN therefore only specifies the order of the UI pages and processing steps, and the data exchanged between them. It models the dialog flow as a transition network, i.e. a directed graph of states connected by transitions called a *dialog graph* (Fig. 4).[2] As illustrated in the communication model above, dialog graphs do not need to be bipartite.

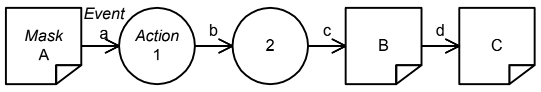

Fig. 4. Dialog graph

The DFN refers to the transitions as *events* and to the states as *dialog elements*, discerning atomic and compound elements. *Atomic dialog elements* are hypertext pages (symbolized by dog-eared sheets and referred to by the more generic term *masks* here) and application logic operations (symbolized by circles and called *actions* from now on). Every dialog element can generate and receive multiple events, enabling the developer to specify much more complex dialog graphs than the linear succession of elements shown above. Which element will receive an event depends both on the event and the generating element (e.g., an event *e* may be received by action *3* if it was generated by mask *D*, but be received by action *4* if generated by mask *E*). Events can carry parameters, i.e. application-specific information such as form input submitted from a mask, and thus facilitate communication between dialog elements.

Theoretically, the complete dialog flow of an application could be described using only atomic elements. However, the resulting specification would be much too complicated to understand, and the "flat" structure does not support reuse of often-needed dialog graphs. The DFN therefore provides *compound dialog elements* (compounds) which encapsulate dialog graphs and realize the key requirement of nested dialog structures: A compound's *interior dialog graph* can contain sub-compounds, and the

[2] The basic concepts and symbols of this notation were inspired by Harel's Statecharts [15], but their semantics have been adapted for the context of hypertext dialog flow specification.

compound itself can be embedded in the *exterior dialog graphs* of super-compounds. We discern two types of compound dialog elements: *Dialog modules* (symbolized by rectangles with rounded corners) contain an interior dialog graph with one entry point and one or more exit points, while *dialog containers* (symbolized by rectangles) contain an interior dialog graph with one entry point, but no exit points.

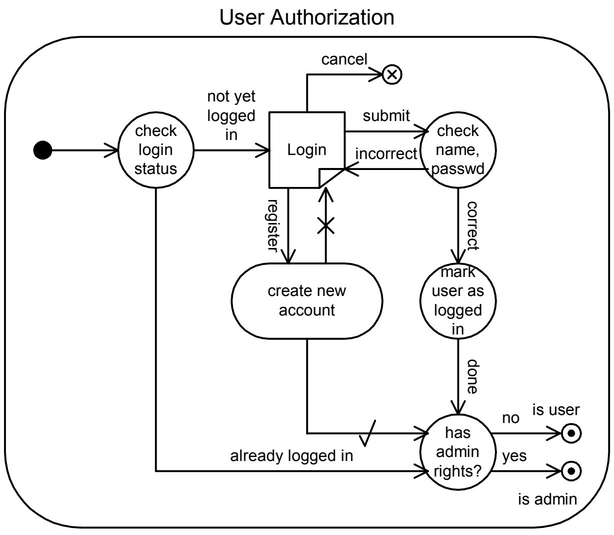

Fig. 5. *User Authorization* dialog module

We will introduce the features of dialog modules using the *User Authorization* module in Fig. 5 as an example. This module checks if the user is already logged in and shows a *Login* mask to prompt for his user name and password, if necessary. If the user's credentials are correct, the module marks him as logged in, checks his access rights and terminates, notifying the super-compound of the user's status. If the user does not yet have an account, he can register using the embedded *create new account* sub-module. Note that by splitting the application logic into relatively fine-grained operations instead of implementing them all in one action, the module can react flexibly to different situations, like bypassing the credential check when the user is already logged in.

Initial and Terminal Events. When a compound receives an event from the exterior dialog graph that it is embedded in, traversal of its interior dialog graph starts with the *initial event*. When the interior dialog graph terminates, it generates a *terminal event* that is propagated to the super-compound and continues the traversal of the exterior dialog graph. Depending on the semantics of the termination, developers can choose between three kinds of terminal events (Table 1):

Table 1. Event types and notation symbols

Event type	**Interior dialog graph symbol**	**Exterior dialog graph symbol**
Initial event	●⟶	*n/a*
Regular terminal event	⊙ Event Name	$\xrightarrow{\text{Event Name}}$
Done terminal event	✓	—✓→
Cancelled terminal event	⊗	—×→
Abort event	×⟶	*n/a*

Regular terminal events are intended to communicate application-specific information to the terminating module's exterior dialog graph, such as the result of an operation or decision (for example, the *User Authorization* module generates an *is user* or *is admin* terminal event, depending on the user's rights). Often, however, modules do not need to notify their calling super-compound about some application-specific state, but should simply indicate if they completed their task successfully or not. The DFN provides the *done* and *cancelled* terminal events to model these situations (for example, the *create new account* module may terminate with a *done* or *cancelled* event, depending on the success of the registration process). In contrast to regular terminal events, *done* and *cancelled* events are unnamed and cannot carry parameters. Their application-independent semantics enable the dialog control logic to handle them automatically in certain situations, as we will see soon.

Compound Events and Return Mechanism. Complex dialog structures will usually contain a certain amount of redundancy, since some dialog elements may be linked from many other elements in the application. If we had to specify all the respective events explicitly, our dialog graph diagrams would soon become cluttered with redundant information. In his Statecharts notation, Harel introduced a special construct to counter the combinatory explosion of transitions that often plagues state machines: a transition leading from a contour to a state [15].

The DFN uses a similar construct, albeit adapted for dialog flow specification: A so-called *compound event*, symbolized in dialog graph diagrams by an arrow leading from the compound's contour to a certain element, indicates that this event may be generated by every element in the compound. As an example, consider the dialog graph of a simple online shop in Fig. 6:[3] The shop's homepage, list of items in each category, detailed description of each item, shopping cart and checkout process shall be linked from every mask in the system. If all events connecting the elements had been specified explicitly, a tangled event web would have been the result. Using compound events, however, we can express the relationships in a much clearer diagram.

[3] The shop was modeled as a dialog container instead of a module since it does not have a natural terminal state.

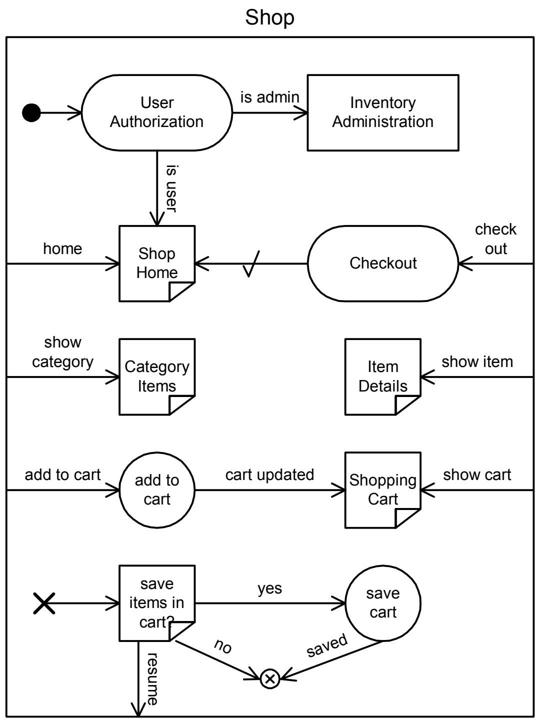

Fig. 6. *Shop* dialog container

Note that the above dialog graph does not explicitly specify what happens when a user does not complete the *Checkout* module with a *done* event, but cancels its execution. For usability reasons, we would not want to return the user to the shop's home page in this case, but to the mask from which he had entered the *Checkout* module (in the same way that window-based applications return the focus to the parent window after the user closed a child window). However, since we do not know at specification time where to return the user, we cannot specify the receiver of the *cancelled* event. The Dialog Control Framework introduced in section 4 solves this apparent dilemma by using the *cancelled* event's application-independent semantics described earlier: If the framework intercepts a *done* or *cancelled* event without a specified receiver, the *return mechanism* automatically leads the event to the dialog mask from which the terminated module was activated, creating the familiar "nesting" effect for the user. Fig. 7 shows a sample dialog sequence employing this mechanism (the gray arrows indicate the compounds' nesting levels).

The scope of compound events only encompasses the compound that they are specified in, but not its super- or sub-compounds. For example, while the *show item* event leads to the *Item Details* mask from any other mask in the *Shop* container, such a connection does not exist for any masks inside the *Checkout* sub-module.

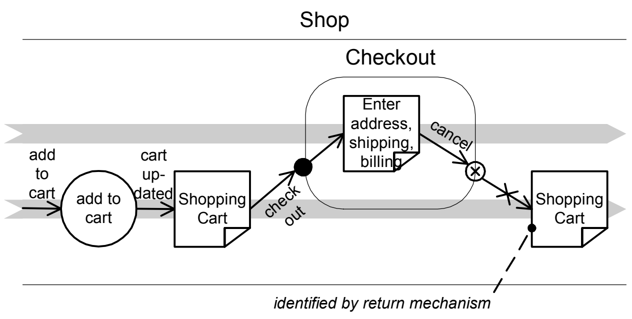

Fig. 7. Return mechanism

Common Events and Abort Mechanism. In some situations, however, it may actually be desirable that certain events can be handled even if their receiver is not specified in the compound that they are generated in – for example, the *create new account* module may be reachable from anywhere within a hypertext-based application, not just from the *Login* mask. To model these relationships, the DFN provides the *common event*. Similar to the compound event, it is symbolized by an arrow leading away from the compound's contour, but outward to another compound element (and only to a compound – it may not lead to an atomic element or into a dialog graph). This so-called *common compound* is nested into the user's dialog sequence wherever he generates the respective common event, independently of his current position in the application.

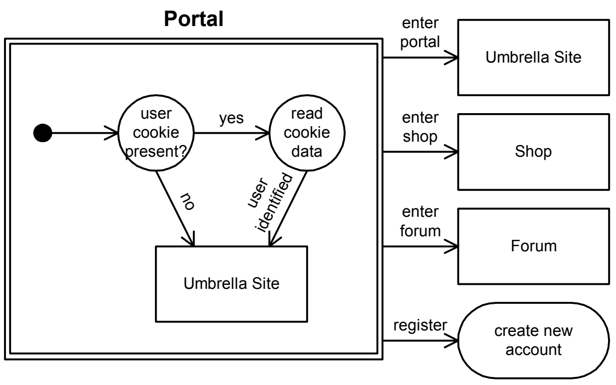

Fig. 8. *Portal* application container

As an example, consider the *Portal* application container in Fig. 8 (the *application container*, symbolized by a double-line box, is the root of the compounds' nesting hierarchy, where every user's dialog sequence starts when he enters the application).

The various parts of this portal system are modeled as common compounds so they can be reached from anywhere within the application.

As with compound events, we need to consider how to return from a common compound. For common modules, we can simply use the return mechanism that leads the user back to the dialog mask that called the module. However, common containers pose more of a challenge. Since they do not terminate by themselves, and nesting them deeper and deeper into each other as the user navigates between them would gradually lock up memory, the only option is to abort a common container before another one can be activated at the same nesting level. For example, if the user is currently in the *Shop* container and generates an *enter portal* event, traversal of the *Shop* container's interior dialog graph (and of all compounds nested into it at the time) has to be aborted before the *Umbrella Site* container's initial event can be handled.

In order to abort a compound in a controlled way, a special *abort dialog graph* can be specified for it, which might ask the user if he really wants to abort (also giving him a chance to resume the original dialog graph where he left off), or if he wants to save any unsaved data before aborting. Traversal of the abort dialog graph, which may not contain any sub-compounds and must not be connected to the compound's regular dialog graph, starts at the *abort event* (see symbol in Table 1) and ends at a *cancelled* terminal event. For example, in the *Shop* container's abort dialog graph (Fig. 6), the system prompts the user if he wants to save the items in his cart before leaving, or if he wants to resume shopping. Fig. 9 shows a dialog sequence using the abort mechanism to switch from the *Shop* to the *Umbrella Site* container.

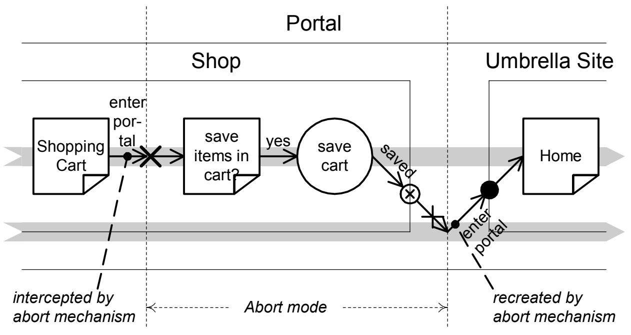

Fig. 9. Abort mechanism

In case the user decides not to switch containers, he can generate a *resume event* (symbolized in dialog graph diagrams by an arrow leading towards the compound's contour), which invokes the *resume mechanism*. Using an algorithm similar to the return mechanism, it leads the user back to the dialog mask in the regular dialog graph that was last displayed before the abort sequence started.

Presentation Channels. The notation constructs introduced so far allow developers to specify complex, hierarchical dialog flows. However, we still need a way to specify the presentation channel-dependent dialog flows required for different client devices, as illustrated in Fig. 2. In the DFN, this can be achieved by specifying the dialog flows for different media in separate dialog compounds and adding the *channel labels* of the respective presentation channels in square brackets after the compound's name.

For example, Fig. 10 specifies the dialog flows for a *Checkout* module on the HTML and WML presentation channel. Note that while the channels employ different dialog masks according to the clients' input/output capabilities, they use the same actions for processing the user's input, as indicated by the shading. This enables developers to implement the device-independent application logic only once and then reuse it for multiple presentation channels. Provided that the actions were designed with sufficient granularity, further channels can be added to an application just by implementing the respective masks and specifying the new channels' dialog flows.

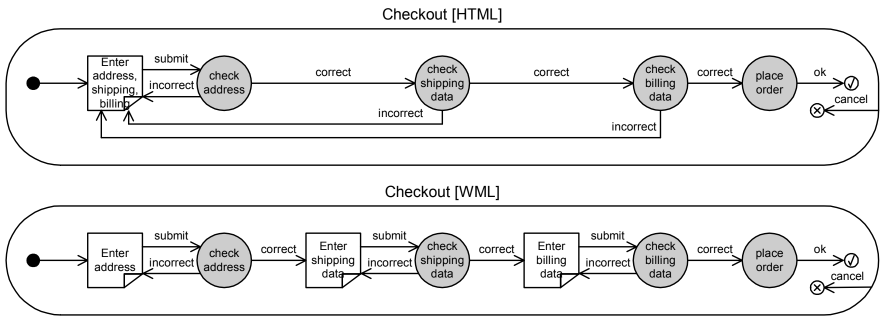

Fig. 10. *Checkout* module on HTML and WML presentation channel

This concludes our presentation of all notation elements. While their semantics were not described formally here, the implementation of the Dialog Control Framework (section 4) defines operational semantics for all constructs.

3.2 Dialog Flow Specification Language

After the dialog flows of an application have been specified in dialog graph diagrams, an efficient transition from specification to implementation is desirable: The dialog graph diagrams should not just visualize the dialog flow, still requiring developers to implement the appropriate dialog control manually, but should rather serve as direct input for the dialog control logic, instructing it how to handle events.

To achieve this, the graphical specifications must first be transformed into a machine-readable representation that can be parsed by the dialog control logic. We therefore introduce the *Dialog Flow Specification Language (DFSL)*, an XML-based language consisting of elements that correspond to the DFN's dialog elements, events and constructs. A complete dialog flow specification consists of two documents – a *dialog flows document* containing a textual representation of the dialog graphs, and a *dialog elements document* mapping dialog elements to their implementation (Fig. 11).

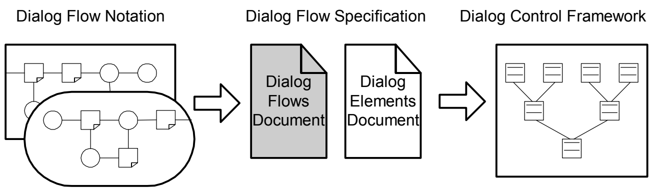

Fig. 11. Transition from specification to implementation

4 Dialog Control Framework

The dialog control logic that reads the DFSL documents and manages the dialog flow accordingly is application-independent. Therefore, we implemented it in a Dialog Control Framework that can be reused for any hypertext-based application and presentation channel. Hypertext-based applications are usually designed according to the Model-View-Controller (MVC) paradigm [18], which suggests the separation of user interface, application logic and control logic. While user interface and application logic can be distinguished quite naturally ("what the user sees" vs. "what the system does"), the distinction between application logic and dialog control logic is much more subtle ("what the system does" vs. "what it should do next, based on the user's input"). Therefore, it is easy to mix up the implementation of application and dialog control logic, even if both are separated well from the presentation logic.

4.1 Struts: Decentralized Dialog Control

For example, in the Apache Jakarta Struts framework [1], the dialog flow is controlled by so-called `Action` objects. Fig. 12 shows how these handle each request:

1. A request comes in from the client.
2. The Controller dispatches the request to the responsible `Action` object, as defined by the action mappings read earlier from a configuration file.
3. The `Action` performs some application logic, either by itself or by calling a subsystem that does the actual work. In the process, the Model data is updated.
4. Based on the outcome of the application logic operation, the `Action` object determines how to proceed in the dialog flow and indicates to the Controller which View should generate the response.
5. The Controller forwards the request to the View indicated by the `Action`.
6. The View generates the response using application data extracted from the Model.
7. The response is sent back to the client.

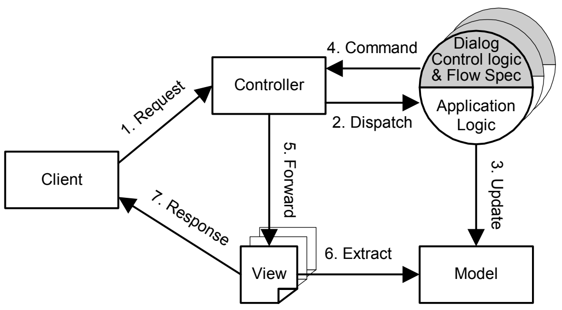

Fig. 12. Coarse architecture of the Struts framework

As indicated by the shading in the figure, the dialog control logic is distributed over all actions in the Struts approach, i.e. the dialog flow is not specified outside the application, but actually implemented in the Java code of the `Action` objects.

This allows the actions to make only relatively isolated dialog flow decisions, and hampers the implementation of more complex dialog structures with constructs like nested dialog modules. To raise the actions' awareness of the "big picture" and enable them to control more complex constructs, still more control logic would have to be implemented in them, exacerbating the problem. Also, the hard-coded decentralized implementation of the dialog control logic is relatively inflexible, almost unsuitable for reuse and hard to maintain. Finally, achieving presentation channel independence would require additional effort and possibly redundant work: Since the dialog flow depends on the presentation channel, while the application logic does not, their close coupling prevents the reuse of actions on multiple presentation channels. Instead, each presentation channel would require its own set of `Action` objects to implement the individual dialog flow for the respective devices.

4.2 DCF: Centralized Dialog Control

In contrast, the Dialog Control Framework (DCF) presented in this paper features a very strict implementation of the MVC pattern, completely separating not only the application logic and user interface, but also the dialog flow specification and dialog control logic: The controller decides where to forward requests by using a central dialog flow model to look up the receivers of events generated by masks and actions [25]. This dialog flow model is an object structure that is not hard-coded anywhere, but constructed automatically from the parsed DFSL documents upon initialization of the framework (Fig. 13).

As the coarse architecture shows, the actions are relatively lightweight here since they contain only application logic, while all dialog control logic has been moved to the *dialog controller*. This controller does not receive requests from the clients directly anymore. Instead, on each presentation channel, it receives events that have been extracted from the requests by *channel servlets*. The dialog controller looks up the receivers for these events in the *dialog flow model* – a collection of objects representing dialog elements that hold references to each other to mirror the dialog flow. This dialog flow model is built upon initialization of the framework by parsing the DFSL documents containing the dialog flow specification (the shaded parts of the diagram emphasize that the dialog control logic and the flow specification are decoupled from the application logic and from each other in this approach). Depending on the receiver that the controller retrieved from the model for an event, it may call an action, forward the request to a mask, nest or terminate compounds. The latter operations are performed on *compound stacks*, which store the nested compounds that constitute the state of the dialog system for each user. We refer to this design pattern as MVC+D (Model-View-Controller plus Dialog Flows).

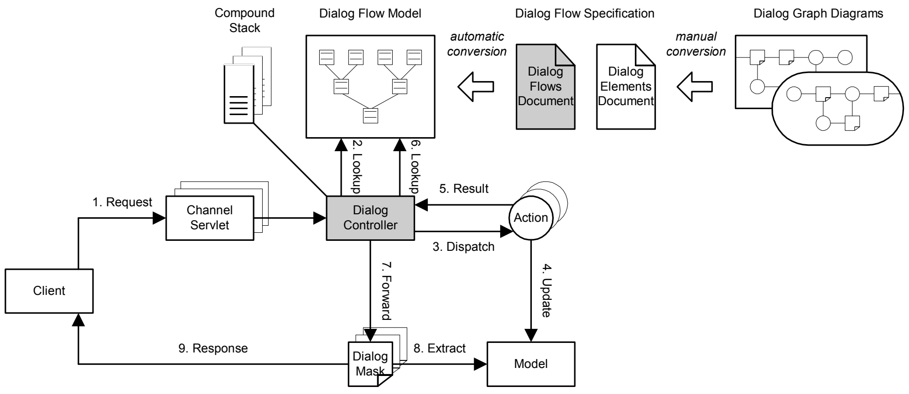

Fig. 13. Coarse architecture of the Dialog Control Framework

In each dialog step, these components work together as follows:

1. A client request with an encoded event is received by a channel servlet, which decodes the event and sends it to the dialog controller.
2. The dialog controller refers to the dialog flow model to look up how to handle this event in the current dialog system state, as stored on the user's compound stack.
3. If an action shall handle the event, it is invoked and the event passed on to it (if a mask shall handle the event, the system proceeds with step 7 instead).
4. The action performs some application logic, either by itself or by calling a subsystem that does the actual work. In the process, the Model data is updated.
5. Based on the outcome of the application logic operation, the action generates a new event and returns it to the dialog controller.
6. The dialog controller refers to the dialog flow model to look up how to handle this event in the current dialog system state, as stored on the user's compound stack.
7. If a mask shall handle the event, the request is forwarded to it (if another action shall handle the event, the system proceeds with step 3 instead).
8. The mask generates the response using application data extracted from the Model.
9. The response is sent back to the client.

For easier comparison with the Struts approach, events involving compounds were not shown in the above sequence. If compounds have to be activated or terminated, the dialog controller would push them onto or retrieve them from the user's compound stack and then look up the next event in the dialog flow model.

This centralized dialog control solution has three advantages over the previously discussed approach:

- The strict separation between application logic implementation, user interface design, dialog flow specification and dialog control logic enables a high degree of flexibility, reusability and maintainability for the components of all four tiers.
- Due to this clean separation, presentation channel-independent applications can be built with minimal redundancy: Only the dialog masks and the dialog flow specifications for the different channels have to be adapted, while the application logic is implemented only once.
- Finally, since the central dialog control logic is aware of the whole dialog flow specified for a channel (it knows the "big picture"), it can provide mechanisms for the realization of complex dialog constructs. Thus, the application developer can use context-independent dialog modules that may be nested, aborted and resumed without having to deal with states, stacks and resume point identification.

To build an application with this framework, the developer does not need to know about the inner structure or implementation of the framework. He only needs to provide subclasses of an `ActionImpl` class implementing the actions, JavaServer Pages implementing the dialog masks, DFSL documents specifying the dialog flow and mapping elements to their implementing entities, and if required, channel servlets for various presentation channels (the prototype framework we implemented already provides `HTMLChannel` and `WMLChannel` servlets). Since these deliverables are completely application-specific, the framework is suitable for black box reuse, giving developers a high degree of flexibility and convenience in building their application.

The authors implemented a prototype of the Dialog Control Framework employing the Java 2 Enterprise Edition. The Dialog Flow Notation elements, events and dialog graph constructs were modeled in a class structure making heavy use of generalization, overwriting and overloading techniques to achieve modularity, extensibility and device independence. To validate the suitability of the Dialog Flow Notation, Dialog Flow Specification Language and Dialog Control Framework for practical use, a demo application that employs all dialog control features was developed at the Chair of Applied Telematics' Mobile Technology Lab. The "Travel Planner" application provides users a front-end for scheduling trips (including reservations for transport and accommodation) that can be accessed via a desktop web browser or a WAP-enabled mobile device. Its development covered all phases from the specification of the dialog flows via their translation into DFSL documents to the framework-based implementation of the application.

5 Conclusions

This paper discussed two challenges brought about by mobile business processes: Firstly, the need to specify the distribution of processes across several sites, and secondly, the need to specify the dialog flows of the applications implementing those processes on mobile devices. It then gave an overview of the Process Landscaping method with its support for refining processes across multiple abstraction layers and associating their activities and objects with distinguished locations. Next, it presented a Dialog Flow Notation and Dialog Control Framework for the specification and management of complex dialog flows in hypertext-based applications.

Introducing the MVC+D pattern, the framework not only strictly distinguishes application logic, user interface and dialog control, but also separates the dialog control logic from the dialog flow specification. The associated notation is essential for providing the specification of the dialog flow to the framework. Since it does not require a detailed knowledge of the underlying protocols and technologies, but instead works with three relatively intuitively understandable concepts ("masks contain what the user sees, actions contain what the system does, and compounds contain transactions the user can perform"), it can also be used by people without programming experience, such as representatives of the application's target audience, usability experts and user interface designers. Therefore, the notation's dialog graph diagrams can be used as a communication tool throughout the software development process. The graphical specifications can be transformed into DFSL documents according to simple rules, allowing for an efficient transition from specification to implementation.

A weak point of the notation might be the fine granularity of actions that is required to employ them flexibly on different presentation channels (this especially concerns actions responsible for processing user input submitted through forms): The finer the actions are grained, the easier it is to adapt to different interaction patterns – however, very fine granularity also results in quite high specification, implementation and performance overhead. When specifying an application, the developer therefore needs to find a balance between the desired flexibility and the required granularity, while being aware that if the granularity is not fine enough, it may be difficult to add more presentation channels to an existing application in the future. Research on solutions to this dilemma is currently in progress.

Another issue that is a current focus of our research is the framework's robustness and error tolerance. When encountering events that cannot be handled, a graceful degradation is the minimum requirement. There are a number of ways in which an event-driven system might react in this case [11], for example by ignoring the event or by reestablishing a clearly defined state. In some situations, however, a more user-friendly reaction would be desirable – most importantly, when the user employs the client's backtracking feature. On the Web, clicking the browser's back button is the second most frequent user activity after clicking on a link [4]. It should therefore not be dismissed as a rare and exceptional activity that can be neglected by the dialog control logic, but rather be regarded as a normal interaction pattern that the application must be able to handle as well as regular clicks on links. Backtracking in a hypertext-based application is similar, but not equivalent to the *undo* feature of traditional applications: While a traditional *undo* aims to reverse a previous application operation, backtracking aims to revisit a previous dialog mask without changing the application's data model. This is a challenge since the user events that are recreated through backtracking often lead to actions, which perform application-logic operations before the dialog step finally completes with displaying a mask.

Finally, more empiric research is needed to see how the Dialog Flow Notation and Dialog Control Framework can be integrated into the software development process for hypertext-based applications. Experiences gained from larger projects should also yield insights into possible limitations of both tools in certain application domains or on certain presentation channels.

References

1. Apache Jakarta Project: Struts. *http://jakarta.apache.org/struts/*
2. Ball, T., Colby, C., Danielsen, P., Jagadeesan, L.J., Jagadeesan, R., Läufer, K., Mataga, P., Rehor, K.: Sisl: Several Interfaces, Single Logic. *International Journal of Speech Technology 3,* 2 (June 2000), 91-106. Kluwer Academic Publishers, 2000
3. Butler, M., Giannetti, F., Gimson, R., Wiley, T.: Device Independence and the Web. *IEEE Computing 6,* 5 (Sep.-Oct. 2002), 81-86
4. Catledge, L.D., Pitkow, J.E.: Characterizing Browsing Strategies in the World Wide Web. *Computer Networks and ISDN Systems 27,* 1065-1073. Elsevier Science, 1995
5. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks 33* (June 2000), 137-157
6. Doberkat, E.E.: A Language for Specifying Hyperdocuments. *Software - Concepts and Tools 17,* 1996, 163-172
7. Fraternali, P., Paolini, P.: Model-Driven Development of Web Applications: The Autoweb System. *ACM Transactions on Information Systems 28,* 4 (Oct. 2000), 323-382
8. Fraternali, P.: Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. *ACM Computing Surveys 31,* 3 (Sep. 1999), 227-263
9. Gaedke, M., Beigl, M., Gellersen, H.-W., Segor, C.: Web Content Delivery to Heterogeneous Mobile Platforms. Advances in Database Technologies, *Lecture Notes in Computer Science 1552.* Springer, 1998
10. Graw, G., Gruhn, V.: Process Management In-the-Many. 4th European Workshop on Software Process Technology (EWSPT '95), *Lecture Notes in Computer Science 913,* 163-178. Springer, 1995
11. Green, M.: A Survey of Three Dialogue Models. *ACM Transactions on Graphics 5,* 3 (July 1986), 244-275
12. Gruhn, V., Wellen, U.: Support for Distributed Business Processes. *Asia-Pacific Software Engineering Conference (APSEC99),* Takamatsu, Japan, December 7-10, 1999
13. Gruhn, V., Wellen, U.: Structuring Complex Software Processes by "Process Landscaping". European Workshop on Software Process Technology (EWSPT 2000), *Lecture Notes in Computer Science 1780.* Springer, 2000
14. Gruhn, V., Wellen, U.: Process Landscaping: Modelling Distributed Processes and Proving Properties of Distributed Process Models. *Lecture Notes in Computer Science 2128,* 103ff. Springer, 2001
15. Harel, D.: Statecharts: A visual formalism for complex systems. *Scientific Computer Programming 8,* 3, 231-274
16. Isakowitz, T., Stohr, E. A., Balasubramanian, P.: RMM: a methodology for structured hypermedia design. *Communications of the ACM 38,* 8 (Aug. 1995), 34-44
17. International Organization for Standardization: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 10: Dialogue principles. *ISO 9241-10,* 1996
18. Krasner, G.E.: A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk. *Journal of Object-Oriented Programming 1,* 3 (1988), 26-49
19. Lupu, E.C., Sloman, M.: Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering 25,* 6, 852-869. IEEE Computer Society Press, 1999

20. Nuseibeh, B., Kramer, J., Finkelstein, A., Leonhardt, U.: Decentralized Process Modelling. 4th European Workshop on Software Process Technology (EWSPT '95), *Lecture Notes in Computer Science 913,* 185-188. Springer, 1995
21. Rice, J., Farquhar, A., Piernot, P, Gruber, T.: Using the web instead of a window system. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '96).* ACM Press, 1996
22. Shaw, M.J.: Electronic Commerce: State of the Art. Shaw, M., Blanning, R., Stader, T., Whinston, A. (eds.): *Handbook on Electronic Commerce,* 3-24. Springer, 2000
23. Sinha, A.: Client-Server Computing. *Communications of the ACM 35,* 7 (July 1992), 77-98
24. Schwabe, D., Rossi, G.: The object-oriented hypermedia design model. *Communications of the ACM 38,* 8 (Aug. 1995), 45-46
25. Singh, I., Stearns, B., Johnson, M., et al.: *Designing Enterprise Applications with the J2EE Platform, 2nd Edition.* Addison-Wesley, 2002
26. Zhao, W., Kearney, D., Gioiosa, G.: Architectures for Web Based Applications. 4th Australasian Workshop on Software and Systems Architectures (AWSA 2002), Feb. 2002, *http://www.dstc.monash.edu.au/awsa2002/papers/Zhao.pdf*