A Dialog Control Framework for Hypertext-Based Applications

Matthias Book, Volker Gruhn
Chair of Applied Telematics/e-Business[*]*, Dept. of Computer Science, University of Leipzig, Germany*
{matthias.book, volker.gruhn}@informatik.uni-leipzig.de

Abstract

Hypertext-based user interfaces have become attractive for many distributed applications today, but they do not reach the usability level of window-based UIs. Because of insufficient dialog control logic, they cannot manage nested and hierarchical dialog structures that users have come to expect from window-based UIs. We therefore present a framework that implements a dialog control logic capable of handling complex, nested dialog structures, and introduce a notation and an XML-based language for specifying such dialog structures. Key concepts are the encapsulation of multiple dialog steps in context-independent dialog modules that can be nested arbitrarily, and the specification of multiple device-specific interaction patterns for a single device-independent application logic. The framework allows black box reuse, leaving only the implementation of the application logic, the design of the user interface and the specification of the dialog flow to application developers.

1. Motivation

Recently, we are observing an increasing trend to implement distributed applications as web-based applications where the user interface (UI) consists of web pages presented in a browser [3]. Common examples are e-commerce applications such as online shops and electronic procurement systems, home banking and customer support systems; groupware applications such as computer supported cooperative work (CSCW), distance learning and community systems; and many other applications such as e-mail readers, remote administration interfaces, portal systems etc.

Most of these web-based applications could also be implemented as "traditional" desktop applications with a window-based graphical UI. However, building such a complex user interface is not trivial (surveys indicate that about half of an application's code volume and implementation effort is spent on the UI [16]) and results in relatively fat clients, which are often undesired in distributed applications where *thin clients* should ideally just be concerned with data presentation, while the server handles all the application logic [18]. Because of their modest requirements regarding client capabilities, hypertext-based applications are suitable for a wide variety of thin client platforms and especially attractive for mobile devices with their strict energy, memory, input and output limitations [10]. Also, hypertext-based UIs can be rendered easily on various presentation channels [4].

However, we need to be aware of a number of software quality issues connected with hypertext-based UIs: Presenting the user interface on pages that are pulled from a server requires a different architecture than letting the application "push" windows on the user's desktop [21]. Since hypertext-based UIs are a relatively new approach, it is very unlikely that the underlying architectures and design patterns have reached the same level of maturity as those for window-based GUIs, especially considering that web-based applications are often developed with a short time to market because of pressure from the market competition [11]. In order to launch an application after a relatively short time, only few resources should be spent on the application-independent control logic, so most resources are available for the business-specific logic.

Judging from experiences with today's hypertext-based applications, sophisticated dialog control logic is often neglected. This is an additional detriment to the usability of hypertext-based applications, which is already suffering from long response times, lack of intuitive interaction techniques and missing interface widgets [17]: Users not only have to work without multiple windows, direct manipulation, right-click context menus and other conveniences that they have come to expect from window-based applications, but are additionally restricted by simplistic dialog structures – 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, leading the user to different pages depending on his credentials, or returning him to where he came from in case of an invalid login)

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

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 those 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, requiring the user to navigate back to the desired area). This violation of the ISO dialog principles of controllability and conformity with user expectations [14] imposes a high cognitive and memory load on the user.

These considerations indicate that nested dialogs could improve the usability of hypertext-based applications. We therefore present the architecture of a Dialog Control Framework capable of managing complex, nested dialog flows on different devices (section 3) and introduce a Dialog Flow Notation for specifying them (section 4). These tools should enable developers to build quality software with sophisticated dialog structures without having to spend resources on the dialog control logic.

2. Related Work

Several frameworks for the implementation of hypertext-based applications 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 [15]. 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 3).

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. Still, Sisl seems more suitable for simple prompt- or menu-based scenarios than for highly interactive applications with complex dialog structures.

A number of notations for the specification of hypertext systems have been proposed over time. However, they mostly focus on *data-intensive* information systems, but not *interaction-intensive* applications [9]: Development processes such as RMM [13] and OOHDM [19], modeling notations and languages such as HDM-lite (used by the Autoweb tool [8]), WebML [6] and DoDL [7] support the generation of web pages out of a large, structured data basis or provide dynamic views on database content, but do not allow the specification of highly interactive features with modular, nested 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 Control Framework and Dialog Flow Notation introduced in the following sections are designed to address these needs.

3. The Dialog Control Framework

Hypertext-based applications are usually designed according to the Model-View-Controller (MVC) paradigm, 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 from the presentation logic.

3.1. Struts: Decentralized Dialog Control

For example, in the Jakarta Struts framework [1], the dialog flow is controlled by so-called `Action` objects. They implement the application logic and also decide where to forward a request, while the controller just executes that forward command (Figure 1).

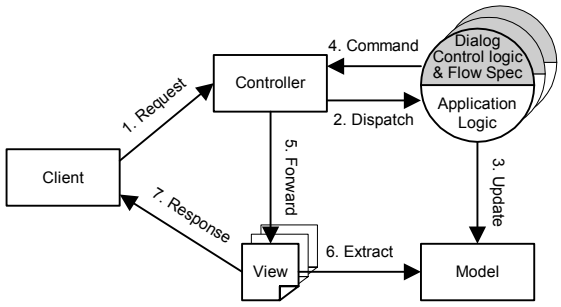

Figure 1. Struts framework architecture

As indicated by the shading in the figure, the dialog control logic is not specified outside the application, but distributed over all actions in the Struts approach. 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.

3.2. 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 actions and views (called dialog masks here) [20]. As the coarse architecture in Figure 2 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 structure is constructed automatically from XML-based dialog flow specification documents upon initialization of the framework. Depending on the receiver that the controller retrieved from the dialog flow model for an event, it may call an action or forward the request to a dialog mask. If dialog modules have to be activated or terminated, the dialog controller pushes them onto or retrieves them from the user's **compound stack**, which stores the nested modules that constitute the state of the dialog system for each user (the concepts of dialog flows, modules and masks will be introduced in section 4). We refer to this design pattern as MVC+D (Model-View-Controller plus Dialog Flows).

This centralized dialog control solution has three advantages over the previously discussed approach: Firstly, 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. Secondly, 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 and the dialog control logic is provided by the framework. 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.

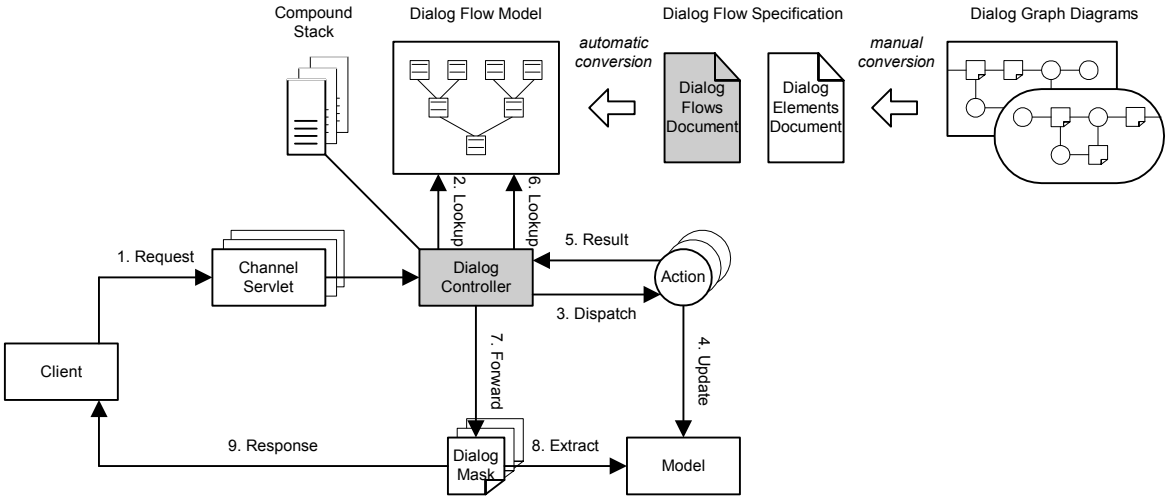

Figure 2. Dialog Control Framework architecture

The authors implemented a reference framework that realizes the functionality described above and provides channel servlets for HTML and WML presentation (further channel servlets for other devices can be added as required by a given application). To build applications with this framework, developers do not need to know about its inner structure or implementation, but can just use it as a black box. They only need to provide subclasses of an `ActionImpl` class implementing the actions, JavaServer Pages implementing the dialog masks, and two documents containing the dialog flow specification and the mapping of elements to their implementations. Written in the XML-based **Dialog Flow Specification Language (DFSL)**, these documents are machine-readable representations of dialog graph diagrams drawn in the Dialog Flow Notation.

4. The Dialog Flow Notation

Inspired by Harel's Statecharts [12][1], the Dialog Flow Notation (DFN) represents the dialog flow within an application as a directed graph of states connected by transitions. To define the concept of a "dialog flow" and develop the notation elements, we first examine the client-server communication taking place in each request-response cycle of a hypertext-based application.

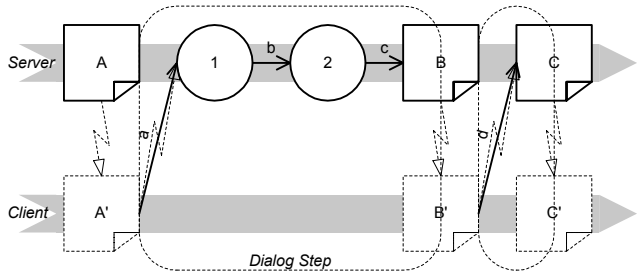

Figure 3. Client-server communication in HTTP

As Figure 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'*).

[1] The semantics of Statecharts' symbols have been adapted for the context of hypertext dialog flow specification in this notation.

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 product information, putting products into the cart, checking out, etc.

4.1. Notation elements

Looking back at the communication model in Figure 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 Dialog Flow Notation 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** (Figure 4). As illustrated in the communication model in Figure 3, dialog graphs do not need to be bipartite.

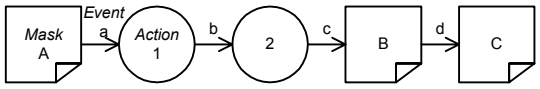

Figure 4. Dialog graph

The notation 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.

4.1.1. Compound dialog elements. Theoretically, the complete dialog flow of an application could be described using only atomic elements. However, the resulting specification would be too complicated to understand, and the "flat" structure does not support reuse of often-needed dialog graphs. The Dialog Flow Notation 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 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 boxes with rounded corners) contain an interior dialog graph with one entry point and one or more exit points, while **dialog containers** (symbolized by boxes with right-angled corners) contain an interior dialog graph with one entry point, but no exit points.

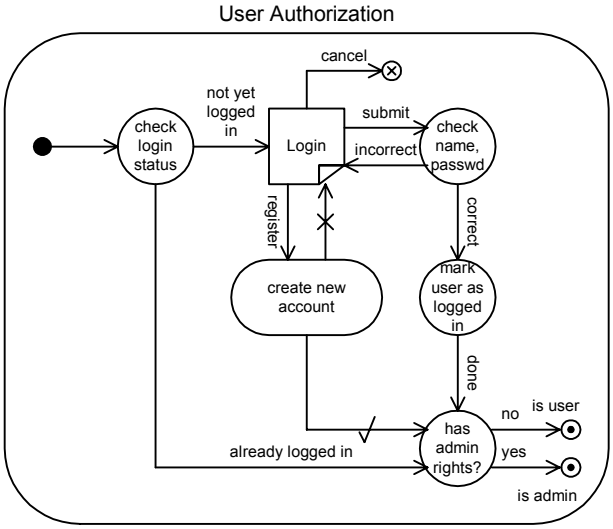

Figure 5. *User Authorization* dialog module

We will introduce the features of dialog modules using the *User Authorization* module in Figure 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 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.

4.1.2. 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	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 Dialog Flow Notation 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 universal, application-independent semantics enable the dialog control logic to handle them automatically in certain situations, as we will see soon.

4.1.3. Compound events. 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 [12].

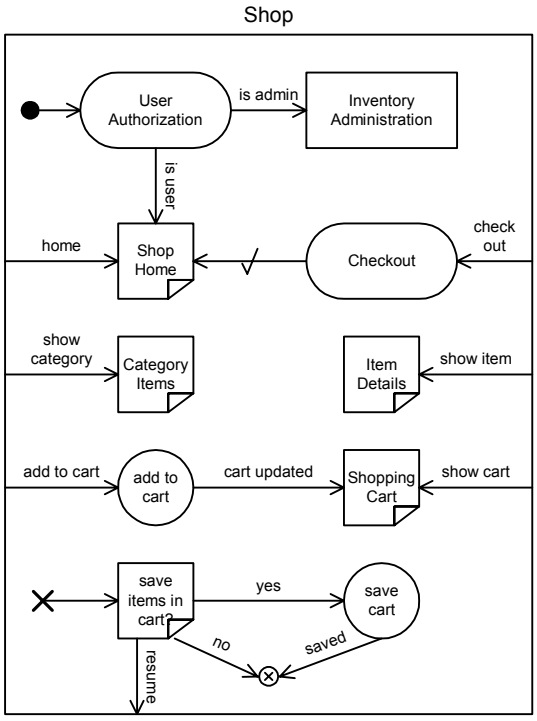

Figure 6. *Shop* dialog container

Our notation 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 Figure 6:[2] 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 same relationships in a much clearer diagram.

4.1.4. Return mechanism. Note that the above dialog graph does not explicitly specify what happens when a user does not complete the checkout process, but aborts it (e.g. by clicking a "cancel" button). 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 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. Figure 7 shows a sample dialog sequence employing this mechanism (the gray arrows indicate the compounds' nesting levels).

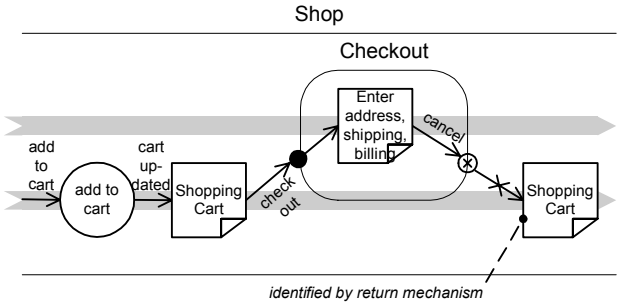

Figure 7. Return mechanism

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.

4.1.5. Common events. 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 Dialog Flow Notation 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.

As an example, consider the *Portal* application container in Figure 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.

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

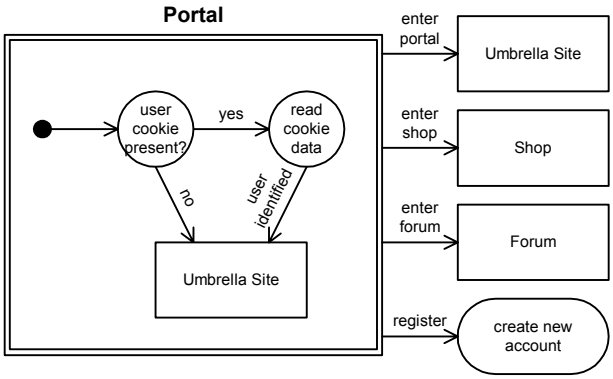

Figure 8. *Portal* application container

4.1.6. Abort and resume mechanism. 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 possibly nested into it at the time) has to be aborted before the *Umbrella Site* container's initial event can be handled.

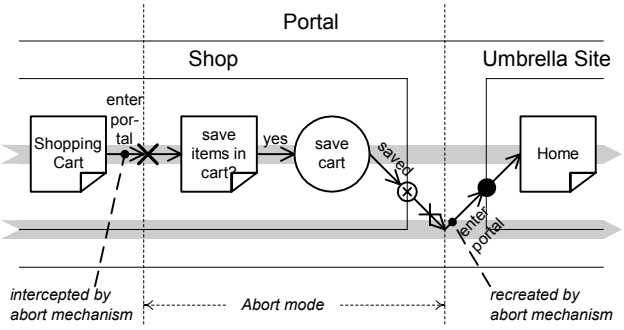

Figure 9. Abort mechanism

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, 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 (Figure 6), the system prompts the user if he wants to save the items in his cart before leaving the shop system, or if he wants to resume shopping. Figure 9 shows an example dialog sequence using the abort mechanism to switch from the *Shop* to the *Umbrella Site* container.

In case the user decides not to switch containers, he can generate a **resume event** (symbolized 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.

4.2. Presentation channels

The notation constructs introduced so far allow developers to specify complex, hierarchical dialog flows. However, we still need a way to specify presentation channel-specific dialog flows for different client devices. In the Dialog Flow Notation, this can be achieved by specifying the dialog flows for different devices in separate dialog compounds and adding **channel labels** after the compound's name. For example, Figure 10 specifies the dialog flows for a *Checkout* module on the HTML and WML 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. This enables developers to implement the device-independent application logic only once and then reuse it for multiple presentation channels.

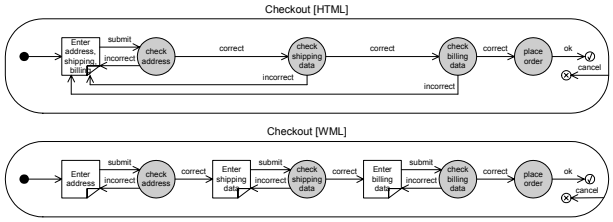

Figure 10. *Checkout* module on HTML and WML presentation channel

5. Conclusions

We presented a Dialog Flow Notation and Dialog Control Framework for the specification and management of dialog flows in web-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, enabling developers to work with nestable dialog modules on different client devices. The dialog flows specified in the graphical notation can be refined incrementally throughout the software development process before being transformed into DFSL documents that serve as input to the framework, enabling a smooth transition from specification to implementation.

While the framework implementation already defines operational semantics for all notation constructs, further work with the notation obviously requires a more solid foundation. We are therefore currently developing formal semantics for the Dialog Flow Notation.

A weak point of the notation may 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, finer granularity also results in higher specification, implementation and performance overhead. Research on solutions to this dilemma is in progress.

Another issue that merits further research is the framework's robustness against backtracking: On the Web, clicking the browser's "back" button is the second most frequent user activity after clicking on a link [5]. It must therefore be regarded as a normal interaction pattern that the application should be able to handle as well as regular clicks on links. 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 that perform application logic operations before the dialog step finally completes with displaying a mask. Research on a mechanism that allows backtracking even through nested dialog modules is currently in progress.

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. Its development covered all phases from the specification of the dialog flows via their translation into DFSL documents to the framework-based implementation. We are striving to gain more empiric experiences from larger projects, which should yield insights into the applicability of the notation and framework in certain application domains and presentation channels, and their integration into various software process models.

6. 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] Baresi, L., Garzotto, F., Paolini, P.: From Web Sites to Web Applications: New Issues for Conceptual Modeling. ER'2000 Workshop on Conceptual Modeling and the Web, *Lecture Notes in Computer Science 1921,* 89-100. Springer, 2000

[4] Butler, M., Giannetti, F., Gimson, R., Wiley, T.: Device Independence and the Web. *IEEE Computing 6,* 5, 81-86

[5] 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

[6] Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks 33,* June 2000, 137-157

[7] Doberkat, E.E.: A Language for Specifying Hyperdocuments. *Software - Concepts and Tools 17,* 1996, 163-172

[8] Fraternali, P., Paolini, P.: Model-Driven Development of Web Applications: The Autoweb System. *ACM Transactions on Information Systems 28,* 4, 323-382

[9] Fraternali, P.: Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. *ACM Computing Surveys 31,* 3 (Sep. 1999), 227-263

[10] 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

[11] Gruhn, V., Schoepe, L., Book, M.: A Specific Software Development Process for an Electronic Commerce Portal. *Proceedings of the 2nd Asia-Pacific Conference on Quality Software (APAQS 2001),* December 10-11, 2001, Hong Kong, China. IEEE Computer Society, 2001, 406-414

[12] Harel, D.: Statecharts: A visual formalism for complex systems. *Scientific Computer Programming 8* (3), 231-274

[13] Isakowitz, T., Stohr, E. A., Balasubramanian, P.: RMM: a methodology for structured hypermedia design. *Communications of the ACM 38,* 8 (Aug. 1995), 34-44

[14] International Organization for Standardization: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 10: Dialogue principles. *ISO 9241-10,* 1996

[15] 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

[16] Myers, B., Rosson, M.B.: Survey on User Interface Programming. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*. ACM Press, 1992

[17] 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

[18] Sinha, A.: Client-Server Computing. *Communications of the ACM 35,* 7 (Jul. 1992), 77-98

[19] Schwabe, D., Rossi, G.: The object-oriented hypermedia design model. *Communications of the ACM 38,* 8, 45-46

[20] Singh, I., Stearns, B., Johnson, M., et al.: *Designing Enterprise Applications with the J2EE Platform, 2nd Edition.* Addison-Wesley, 2002

[21] Zhao, W., Kearney, D., Gioiosa, G.: Architectures for Web Based Applications. 4th Australasian Workshop on Software and Systems Architectures (AWSA 2002), *http://www.dstc.monash.edu.au/awsa2002/papers/Zhao.pdf*