

# Efficient Modeling of Hierarchical Dialog Flows for Multi-Channel Web Applications

Matthias Book, Volker Gruhn

Deutsche Telekom Chair of Applied Telematics/e-Business, University of Leipzig  
Klostergasse 3, 04109 Leipzig, Germany; Tel. +49-341-97-32337, Fax +49-341-97-32339  
{book, gruhn}@ebus.informatik.uni-leipzig.de

## Abstract

*In web-based applications, most user interactions take the form of navigating between web pages. The structure of the navigation model thus has a strong impact on a web application's usability. However, specifying a user-friendly navigation model for complex applications can be time-consuming, especially when designing for multiple presentation channels. We therefore present the formal semantics of the Dialog Flow Notation (DFN) that provides constructs for the design of modular navigation models, and especially focus on constructs that reduce the specification redundancy within and between channels, thus reducing the design effort for web-based user interfaces.*

## 1. Introduction

The ISO dialog principles “suitability for the task” and “conformity with user expectations” state that user interfaces should be designed in a way that reflects and supports the way in which users would naturally handle a problem, but not in a way that is determined by technical aspects of the system's implementation [14]. In the development of web-based applications, this can be a challenge since hypertext offers much less intrinsic support for complex widgets and interaction patterns than window-based applications do. Instead, most user interactions take the form of navigating between web pages. The structure of a web application's navigation model (or *dialog flow*, as we call it) thus is a primary determinant of the application's usability.

However, due to technical limitations of the Web and the terminal devices (e.g. page-based user interface, pull communication, limited input and output capabilities), the implementation and use of web applications is a bigger challenge than that posed by traditional window-based applications. This is apparent in two major aspects: Firstly, users are used to work with hierarchical dialog structures (visualized by overlapping dialog boxes in window-based applica-

tions), while the Web only allows flat dialog structures without additional dialog control support [17]. Secondly, different terminal devices may require different interaction patterns in order to complete the same business process, since complex dialog masks may have to be split up into several pages instead of being presented to the user as a whole [5]. If web applications do not address these challenges, their usability suffers.

We argue that the dialog flow should therefore be the driving aspect of the whole development process. In order to fulfill this role, the dialog flow needs to be intuitively and efficiently modeled and refined in accordance with requirements. The notation used to do this should have sufficient expressive power to describe interaction patterns that the user has become accustomed to through window-based applications (such as nesting tasks by stacking windows on top of each other), be flexible enough to be adapted to different presentation channels (e.g. various rendering devices) without requiring redundant specifications, and be suitable for automatic transformation into an executable format to avoid manual reimplementation of the specifications.

In this paper, we show how our Dialog Flow Notation (DFN) [3] fulfills these requirements. The DFN models a web-based application's dialog flow as a transition network, i.e. a directed graph of states connected by transitions called a *dialog graph*. We refer to the transitions as *events* and to the states as *dialog elements*. Two of the most important dialog elements are *masks* (hypertext pages symbolized by dog-eared sheets) and *actions* (application logic operations symbolized by circles). Every dialog element can generate and receive multiple events (symbolized by arrows). 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  $A_1$  if it was generated by mask  $M_1$ , but be received by action  $A_2$  if generated by mask  $M_2$ ). Events can carry parameters such as form data or processing results, and thus facilitate communication between dialog elements.

In the DFN, dialog graphs are never free-standing, but always encapsulated into *dialog modules* that facilitate the

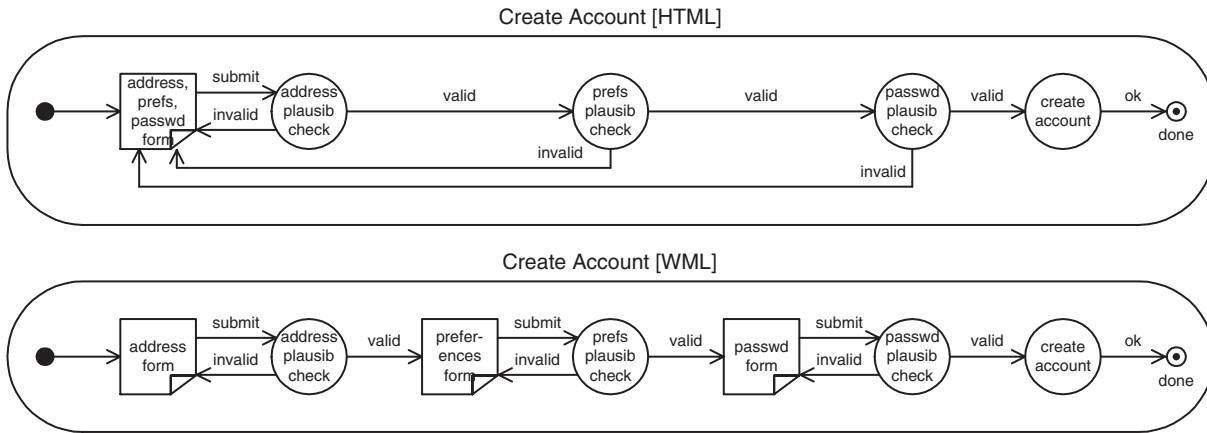


Figure 1. Dialog graphs for “Create Account” module on HTML and WML presentation channel

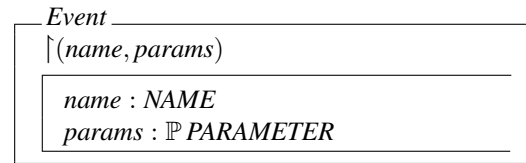
connection and nesting of dialog graphs, as illustrated in Fig. 1 using the example of a “Create Account” module. A module typically encapsulates one or more dialog masks, actions and possibly sub-modules implementing a certain functionality, process or behavior in the system (e.g. creating an account, logging in, searching for hotels, booking a room etc. in a travel portal). Any module’s dialog graph can contain sub-modules, and any module can itself be embedded in the dialog graphs of various super-modules. The modules’ definitions are decoupled from each other, but through the interfaces of their initial and terminal events, they can call and return results to each other. This facilitates easy reuse of dialog sequences and enables developers to model the complete dialog flow of an application with a set of dialog modules that are nested into and connected with each other. To model different interaction patterns that may be required by different devices, developers can specify variants of a module’s dialog flow by assigning *presentation channel* labels to them (noted in square brackets after the module name). The dialog graphs modeled in the DFN can be transformed into XML specifications that serve as input for our Dialog Control Framework (DCF) (Sect. 4).

In the following sections, we will describe the formal semantics of the notation elements and their relationships in Object-Z [6]. While the full DFN comprises more elements than those shown here, we restrict our presentation to those constructs that minimize redundancy in the dialog graphs and thus increase the efficiency of the specification effort. We first address the static aspects (i.e. how dialog graphs are specified in the DFN) in Sect. 2 and then present the dynamic aspects (i.e. how the dialog graph specifications are interpreted by the dialog control logic) in Sect. 3. As an example for a concrete implementation of these formal specifications, we present the architecture of the DCF in Sect. 4. After an overview of the related work (Sect. 5), we conclude with an outlook on further research in Sect. 6.

## 2. Static Aspects: Dialog Graph Specification

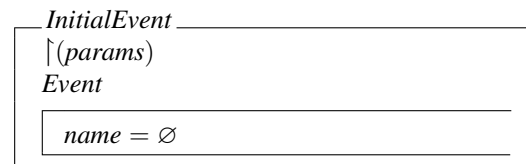
### 2.1. Dialog Events

Dialog events are the core construct of the Dialog Flow Notation, as they connect all other elements. Each *Event* is characterized by a name and a set of parameters:



Since the parameters do not carry any information that is relevant for dialog control, but serve only for the conveyance of application-specific data between dialog elements, we will just use the basic type  $[PARAMETER]$  for them and leave their actual structure open to concrete implementations. The names of the events will typically be strings; we simply use the basic type  $[NAME]$  here.

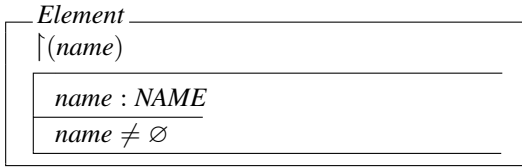
Most events connecting the elements in dialog graphs are “regular” events such as the one above. However, to indicate the first dialog element that shall be invoked when a dialog module is entered, we need a special *InitialEvent*:



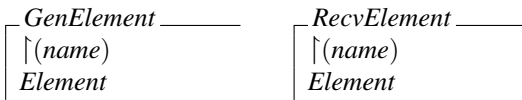
As a sub-class of *Event*, it can also carry parameters (comparable to arguments in a method call), but is unnamed since it is never called explicitly, but always traversed implicitly upon entering a dialog module. Since we’ll occasionally need an initial event instance later on, we already define the global variable  $initialEvent : InitialEvent$  here.

## 2.2. Dialog Elements

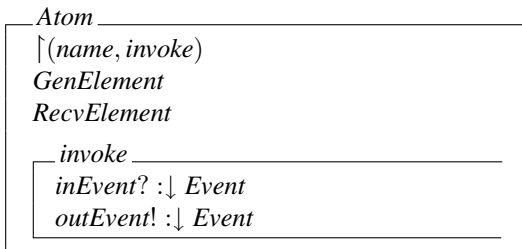
Dialog elements are the second core concept in the Dialog Flow Notation. They represent the entities that users can navigate among using the events. Every *Element* is characterized by a name:



Before we introduce the actual dialog elements, we derive two “abstract” marker sub-classes from *Element* that do not add any declarations, but will help us distinguish element types that can generate events and types that can receive events later on:

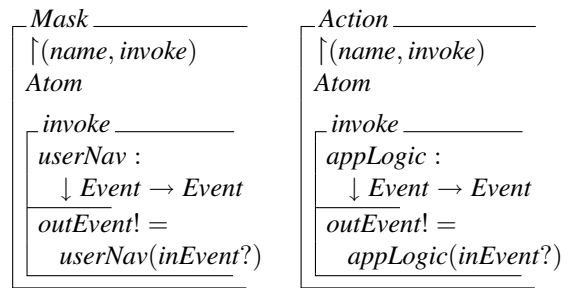


Most elements in a dialog graph either represent a dialog mask (i.e. a page displayed to the user, e.g. “address form” in Fig. 1) or an action (i.e. operations executed by the application logic, e.g. “address plausib check” in Fig. 1). These atomic dialog elements can both receive an event and generate a new event when invoked:

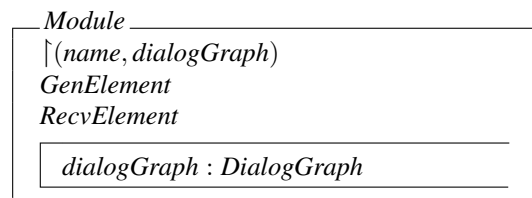


From this “abstract” super-class *Atom*, the two concrete dialog elements *Mask* and *Action* are derived. They differ only in the way that they generate a new outgoing event: While a *Mask* uses the parameters in the incoming event to render a web page and let the user generate the outgoing event by clicking on a link (represented by the *userNav* function), an *Action* feeds the parameters of the incoming event into a piece of application logic that will determine the outgoing event (represented by the *appLogic* function):<sup>1</sup>

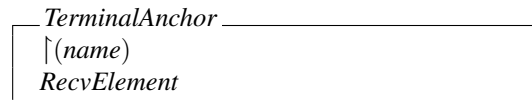
<sup>1</sup> Note that by using the type declaration  $\downarrow Event$  for the function’s argument, we also allow the *InitialEvent* sub-class as an incoming event. However, since dialog elements cannot generate initial events themselves, only the regular *Event* type is declared as the function result. Constructs like this are used throughout the specification to restrict or allow certain event or element types.



In contrast to atomic elements, so-called dialog modules can not only be embedded into a dialog graph, but also contain dialog graphs of their own (such as the “Create Account” module in Fig. 1). A *Module* can receive events, which will lead to the traversal of its interior dialog graph, and generate events that are created when their interior dialog graph terminates. They do not need an *invoke* operation since their invocation and termination is handled by the *DialogController* (Sect. 3):



Before we go into detail on the structure of the *dialogGraph*, we need to define a last dialog element type: *TerminalAnchors* (such as the “done” anchor in Fig. 1) are used inside a module to indicate that it should be terminated when traversal of the dialog graph reaches this point. The terminal anchor can therefore only receive events, but not generate them:



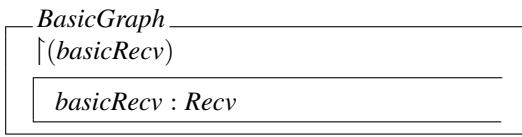
After termination of a module, traversal of the dialog graph that it was embedded in will be continued with an event carrying the name of the respective terminal anchor, as described in Sect. 3.

## 2.3. Dialog Graphs

In the DFN, all dialog graphs are contained in dialog modules. At the core of each dialog graph is a function *Recv* that indicates which elements will receive which events generated by which other elements:

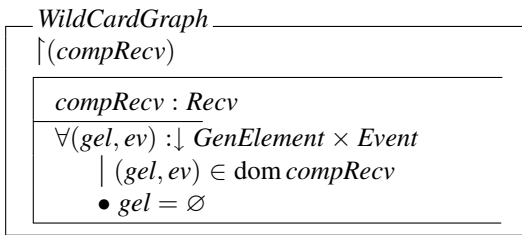
$$Recv == \downarrow GenElement \times \downarrow Event \rightarrow \downarrow RecvElement$$

A dialog graph can thus basically be defined as a class containing such a receiver function:

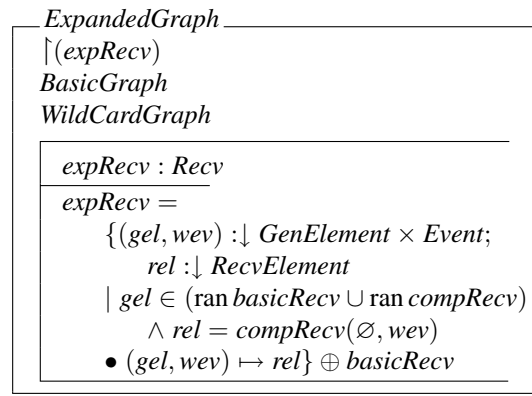


While this **BasicGraph** already has all the expressive power required to describe complex, modular dialog flows, doing so is not efficient in practice because dialog graphs typically contain quite a bit of redundancy: For one thing, some central elements (such as the home page) are typically reachable from almost every page, but spelling all those links out in the dialog graph specification would be too cumbersome. For another thing, dialog graph variants for different presentation channels are often similar in some aspects (since they are intended for the same task), while differing in other aspects (due to factors such as smaller display resolution that require more dialog steps to accomplish the same task). Ideally, this redundancy between channels should also be avoided in the specification.

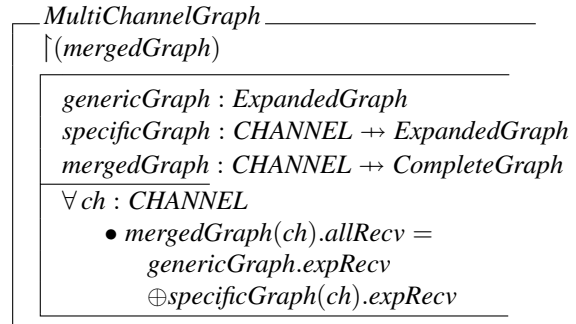
The DFN therefore provides a couple of additional constructs for specifying complex multi-channel dialog graphs with low redundancy. First of all, it allows the specification of so-called “compound events” that do not require the specification of a generating event. They can be bundled in a **WildCardGraph**, which is characterized by the fact that the generating elements  $gel$  of all events in its  $compRecv$  function are unspecified ( $\emptyset$ ):



Compound events carry the semantics that they may be generated by any element in the same dialog graph, without the developer having to explicitly specify all those mappings (so specifying one compound event leading to the home page is sufficient to indicate that the home page can be reached from all elements in the same module). The dialog graph implicitly defined this way is described by the **ExpandedGraph** schema. In the  $expRecv$  function, it introduces an event  $wev$  leading from every generating element  $gel$  that is in the same module (i.e. reachable through a regular or compound event;  $gel \in (\text{ran } \text{basicRecv} \cup \text{ran } \text{compRecv})$ ) to every element  $rel$  that receives the respective compound event  $wev$  (i.e.  $rel = \text{compRecv}(\emptyset, wev)$ ). The set of receiver mappings  $(gel, wev) \mapsto rel$  defined in this way is overridden by the receiver mappings in the  $basicRecv$  function in order to ensure that an explicitly specified event takes precedence over a compound event with the same name.

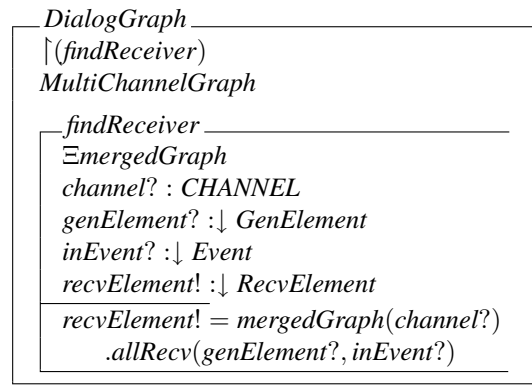
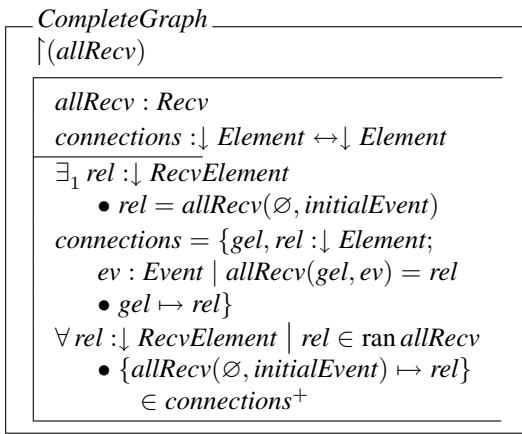


Compound events allow us to increase the specification’s efficiency within the dialog graph of one channel. However, as mentioned above, additional redundancy may exist between the dialog graphs of different presentation channels. In order to reduce this redundancy, the DFN provides the concept of generic and channel-specific dialog graphs: All events that are specified in the generic graph are implicitly specified for all channels, while those events that are specified in a channel-specific graph are only valid for that particular channel. These semantics are expressed by the **MultiChannelGraph** class, which contains one  $genericGraph$ , and the function  $specificGraph$ , which maps the various channel labels to their respective specific graphs (assuming the DFN’s presentation channel labels are included in a free type definition like  $\text{CHANNEL} ::= \text{html} \mid \text{wml} \mid \dots$ ):



This approach to the specification of multi-channel dialog graphs reduces redundancy since it allows developers to specify the fragments of the dialog flow that are common to all channels in the  $genericGraph$ , and the fragments that are unique to a certain channel  $ch$  in its respective  $specificGraph(ch)$ . We then arrive at the complete dialog graph for a channel  $ch$  by merging the generic and specific graphs, where events on the specific graph override events with the same name on the generic graph ( $\text{mergedGraph}(ch).allRecv = \text{genericGraph}.expRecv \oplus \text{specificGraph}(ch).expRecv$ ).

The complete dialog graph that results from this operation for each channel is described by the class **CompleteGraph**:



### 3. Dynamic Aspects: Dialog Control Logic

This schema declares two important constraints that the final receiver mapping *allRecv* needs to fulfill: Firstly, the dialog graph must contain exactly one initial event receiver ( $\exists_1 \text{rel} : \downarrow \text{RecvElement} \bullet \text{rel} = \text{allRecv}(\emptyset, \text{initialEvent})$ ).

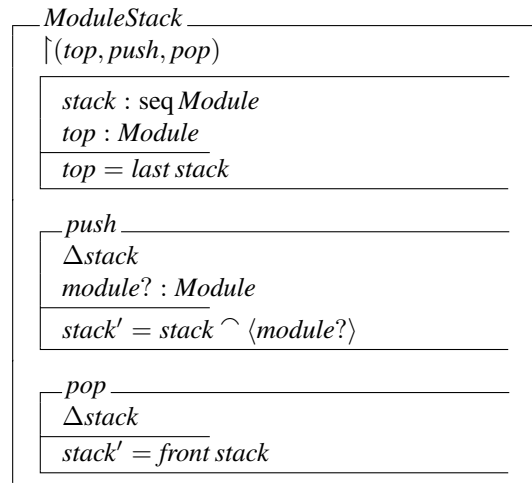
Secondly, the dialog graph must take the form of an arborescence, i.e. all elements must be reachable by following a path of events starting from the initial event. We formulate this constraint by defining the relation *connections* that maps an element *gel* to an element *rel* iff there is an event *ev* leading from *gel* to *rel* (i.e.  $\text{allRecv}(\text{gel}, \text{ev}) = \text{rel}$ ). Then, we demand that its transitive closure  $\text{connections}^+$  must contain mappings between the first element in the dialog graph (i.e. the one receiving the initial event;  $\text{allRecv}(\emptyset, \text{initialEvent})$ ) and every other element *rel* in the graph ( $\text{rel} \in \text{ran allRecv}$ ).

Note that these constraints could not be formulated for the generic and specific graphs because those are fragmentary – for example, a missing initial event in a channel-specific graph is admissible if the generic graph provides one. However, we can formulate these constraints now after merging the graphs. Also, note that the *CompleteGraph* does not have to be manually specified, but can be calculated from the simple *BasicGraphs* and *WildCardGraphs* specified initially for each channel.

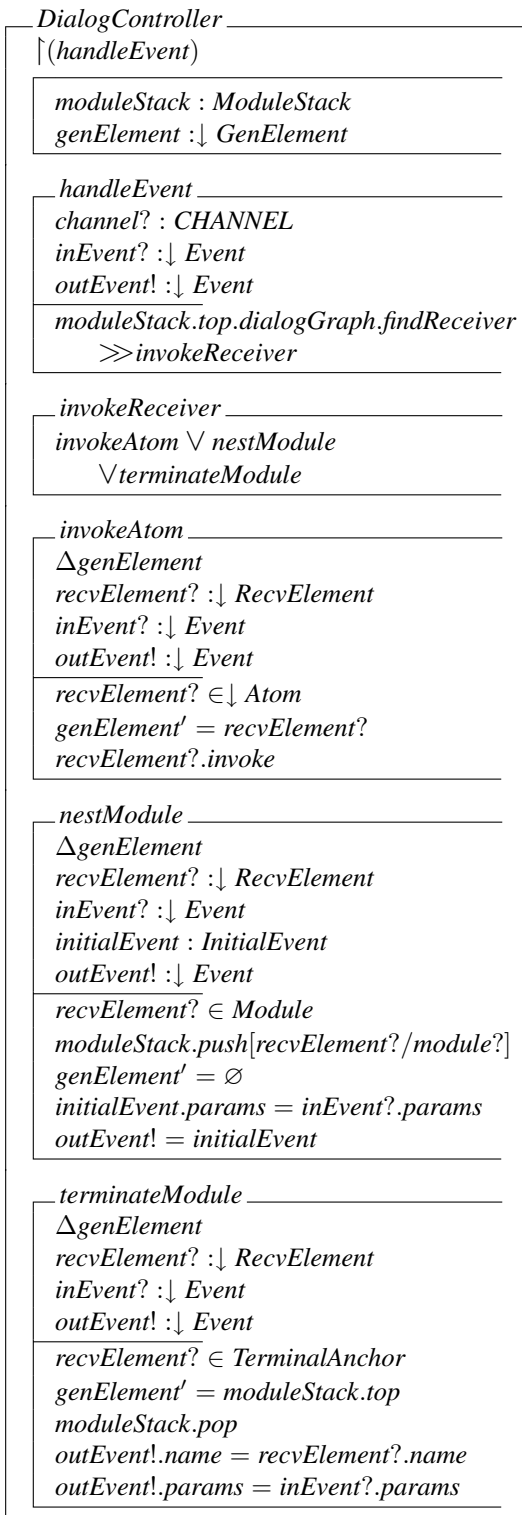
Having thus specified multi-channel dialog graphs and the constraints on them, we finally need to declare how they are embedded in dialog modules and how the receiver for a particular event is actually found. The class **DialogGraph** fulfills this purpose – it is derived from *MultiChannelGraph*, but encapsulates the actual receiver mappings and makes only the *findReceiver* operation visible to the outside. This operation returns the receiving element *recvElement!* for the event *inEvent?* coming in from the generating element *genElement?* by feeding these arguments into the *allRecv* function of the *mergedGraph* on the presentation channel *channel?* employed by the user. Each *Module* contains one of these dialog graphs in its *dialogGraph* attribute:

The schemas presented in the previous section define the semantics of the various element and event types introduced by the DFN and show the constraints that dialog graphs constructed from them need to satisfy. In this section, we will define the dynamic aspects of the dialog control logic, i.e. we show how the dialog controller shall react to incoming events and what happens when the various kinds of dialog elements are encountered in the dialog graph.

Since the DFN allows the nesting of dialog modules, we first need a data structure that contains the “call stack” that is created at run-time when one module is invoked within another. For this purpose, the **ModuleStack** class contains a sequence of *Modules*. The *top* attribute always refers to the last element in this sequence (i.e. the top module on the stack), and the operation *push* appends the given module *module?* to the sequence, while the operation *pop* reduces the sequence to all but the last element, thereby removing the top module from the stack:



The actual dialog control logic is contained in the **DialogController** class:



This class encapsulates the *moduleStack* and a *genElement* attribute that always keeps track of the element that was invoked in the previous dialog step (we'll see how this is ensured shortly). In order to handle incoming events, the dialog controller provides the *handleEvent*

operation, which expects the channel that the user is working on and the incoming event as arguments. Using these, the top module's *findReceiver* operation can identify the receiving element, as described in Sect. 2.3. The found receiving element is then invoked (expressed in Object-Z by piping *findReceiver*'s output *recvElement!* into *invokeReceiver*'s argument *recvElement?*). Since the *invokeReceiver* operation is a schema disjunction, the actual operation depends on the receiving element's type, as expressed by the pre-conditions of those operations:

If the receiving element is a mask or an action (i.e. an atomic dialog element; *recvElement?*  $\in$   $\downarrow$  *Atom*), the *invokeAtom* operation declares this receiving element as the next generating element, and then calls its *invoke* operation, which will actually determine the new *outEvent!*, as described in Sect. 2.2.

If the receiving element is a dialog module (*recvElement?*  $\in$  *Module*), the *nestModule* operation first pushes this module onto the module stack (we need to rename the argument to make it compatible to the *push* interface). Then, it clears the generating element attribute (*genElement* =  $\emptyset$ ) since we need to start the module's interior dialog graph with an initial event, which by definition does not have a generating element (as it is generated by the dialog controller). Finally, the incoming event's parameters are copied to a new initial event that becomes the operation's outgoing event.

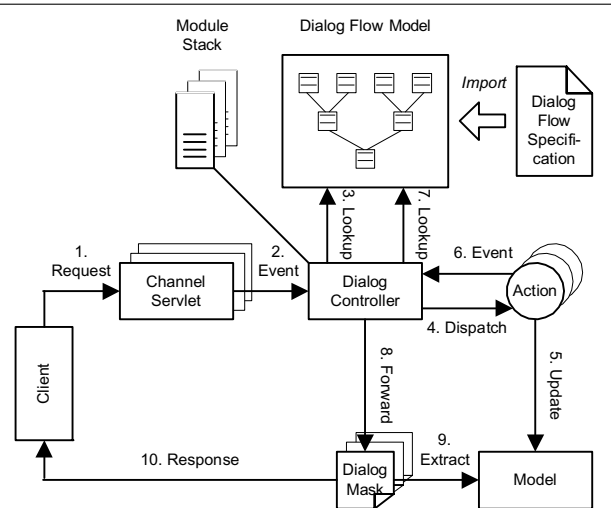
Last but not least, if the receiving element is a terminal anchor (*recvElement?*  $\in$  *TerminalAnchor*), the *terminateModule* operation first declares the module that's still on top of the stack as the generating element of the next event, and then removes it from the stack. Since the name of the terminal anchor always indicates the name of the outgoing event that will continue the terminated module's exterior dialog graph, we copy the *recvElement?*'s *name* to the outgoing event's name, and finally copy the parameters of the event that lead to the module's termination into its outgoing event.

The newly generated *outEvent!* can now again be handled by the *handleEvent* operation, and the cycle repeats.

## 4. Concrete Implementation

This concludes our description of the static and dynamic aspects of dialog graph specification and interpretation in the DFN. For a concrete implementation of this formal specification, we suggest the architecture in Fig. 2, which is used by our Dialog Control Framework (DCF) [3].

Upon initialization of the application, the DCF parses the dialog flow specification and builds an object-oriented dialog flow model from it. Every time an event comes in from a client through one of the presentation channel servlets (steps 1 and 2), the dialog controller looks up the receiver



**Figure 2. Dialog Control Framework**

for this event in the dialog flow model (3). If the receiver is an action, the dialog controller will dispatch the event to the respective object (4), which may update the applications' data model (5) and then returns a new event indicating the result of the operation (6). The dialog controller again looks this event up in the dialog flow model (7), where it may find that it leads to another action (in which case the cycle repeats) or to a module or mask. If the event receiver is a module, the dialog controller will push a reference to it onto the user's module stack in order to reflect the user's updated position in the dialog flow, and then look up the receiver of that module's initial event. If the event receiver is a mask, the dialog controller will dispatch the request to the implementation for the corresponding presentation channel (e.g. a JavaServer Page, step 8), which can read information from the data model (9) to build a response that is finally sent back to the client (10).

Based on the MVC pattern [15], this approach enforces a strict separation of presentation, application and dialog control logic since neither the masks nor the actions determine the next step in the dialog flow directly. Instead, this decision is made by the dialog controller according to the dialog flow specification.

## 5. Related Work

Other notations suggested for modeling web-based UIs initially focused on data-intensive information systems, but not interaction-intensive applications [9]: For example, the RMM development process [13] allows the definition of navigable relationships between data entities, and the OOHDM [19] process provides classes like node, link and index to represent different forms of navigation; however, they do not provide explicit support

for modeling different presentation channels. The language WebML [7] is capable of modeling the layout and appearance of web pages independently of the output device using an abstract XML language for its presentation model, but does not seem to provide an overriding mechanism for the extension of generic dialog modules with channel-specific fragments that enables the easy reuse of dialog sequences across presentation channels. Similarly, the Web Composition Language [10] focuses on the specification of the dialog mask's contents, but seems to lack tool support for handling navigation patterns.

More recent notations, often based on Statecharts [12], also provide extensive support for interaction-intensive applications: For example, Leung et al. [16] use Statecharts to model dynamic web applications, but do not provide a means for specifying device-specific interaction patterns. The same is true for StateWebCharts [20], and the HMBS model [8] focuses more on challenges such as synchronization that are introduced by multimedia elements embedded into hypertext. Schewe et al. [18] 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 a means for the non-redundant specification of device-independent dialog flows. Last but not least, the formal model for web interactions proposed by Graunke et al. [11] helps to identify and deal with unexpected situations in the dialog flow (e.g. backtracking), but also does not feature dialog control support through a compatible framework.

Most tools offering dialog control implementation support for web applications follow the Front Controller design pattern to facilitate easier dialog control. However, since they lack accompanying notations, they still require developers to manually implement dialog flows that were specified using unrelated notations (if at all). The Apache Jakarta Project's Struts framework [1] is the most popular solution today, however, it forces developers to combine application logic and dialog control logic in its actions: The Struts controller only decides which action should receive incoming requests, but the actions then decide which view to display next. Since the application logic is thus not completely decoupled from the dialog flow, reusing it on different channels is not always possible.

The challenges posed by different devices' interaction patterns are addressed in the Sisl (Several Interfaces, Single Logic) approach [2]. Its "service monitor" can process unordered or incomplete input from a wide range of client devices. However, since it uses acyclic graphs to model dialogs, it is more suitable for simple linear and branched dialog structures than for highly interactive applications with nested and cyclic dialogs. The need to spread complex forms over multiple interaction steps on small-screen devices instead of presenting them as a whole is addressed by

the Renderer-Independent Markup Language (RIML) [21], an extension of XHTML 2.0 which contains semantic information for an automatic pagination engine. Collecting the data fragments coming in from the split-up forms is the task of a proxy between the client and server in that approach. In contrast, we are working on an extension to the DCF to enable it to manage the necessary micro-dialog flows directly.

## 6. Conclusions

In the preceding sections, we presented the formal semantics of our Dialog Flow Notation (DFN), which enables the specification of modular dialog flows for multi-channel web applications with low redundancy. This is achieved by specifying events that can be generated by every element only once as compound events (instead of explicitly spelling out all possible links), and by placing the shared dialog graph fragments of all presentation channels in one generic channel, so only the channel-specific fragments have to be specified explicitly for each channel. These savings result in increased specification efficiency. By defining the static and dynamic aspects of the notation in Object-Z, we could present finer points such as rules for overriding events and the order of merging vs. overriding operations unambiguously, as opposed to a prose description of those constructs.

Our prototypic implementation of the travel portal AR-GuS, a complex web application, already yielded encouraging results regarding the savings in implementation effort that seem possible using this approach [4]. In our ongoing research, we are striving to gain more insight into the impact that the DFN and DCF have on the development effort of web-based applications. At the same time, we are working on extensions to the DCF that will allow it to generate dialog graphs for different presentation channels automatically, and on algorithms for the identification and handling of unexpected client-side navigation (i.e. users clicking the back button, cloning windows, etc.). We hypothesize that these extensions should increase developer's efficiency further, while at the same time increasing web applications' usability.

## References

- [1] Apache Software Foundation. Struts. <http://struts.apache.org>
- [2] T. Ball, C. Colby, and P. Danielsen. Sisl: Several interfaces, single logic. *Intl J Speech Technology*, 3(2):91–106, 2000.
- [3] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *19th IEEE Intl Conf Automated Software Engineering (ASE 2004)*, pages 100–109. IEEE Computer Society Press, 2004.
- [4] M. Book and V. Gruhn. Experiences with a dialog-driven process model for web application development. In *COMP-SAC 2005 Workshop on Model Driven Agile Development II - Component-Based Agile Development*, pages 173–178. IEEE Computer Society Press, 2005.
- [5] M. Butler, F. Giannetti, R. Gimson, and T. Wiley. Device independence and the web. *IEEE Computing*, 6(5):81–86, Sep–Oct 2002.
- [6] D. A. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In *FORTE '89: Proc IFIP TC/WG6.1 2nd Intl Conf Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 281–296. North-Holland, 1990.
- [7] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33:137–157, 2000.
- [8] M. C. F. de Oliveira, M. A. S. Turine, and P. C. Masiero. A statechart-based model for hypermedia applications. *ACM Trans Information Systems*, 19(1):28–52, 2001.
- [9] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, Sep 1999.
- [10] M. Gaedke, M. Beigl, H. Gellersen, and C. Segor. Web content delivery to heterogeneous mobile platforms. *Advances in Database Technologies, LNCS*, 1552, 1998.
- [11] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. *Proc 12th European Symposium on Programming, LNCS*, 2618, 2003.
- [12] D. Harel. Statecharts: A visual formalism for complex systems. *Science Comp Programming*, 8(3):231–274, 1987.
- [13] T. Isakowitz, E. Stohr, and P. Balasubramanian. RMM: a methodology for structured hypermedia design. *Comm ACM*, 38(8):34–44, Aug 1995.
- [14] Intl Organization for Standardization. Ergonomic requirements for office work with visual display terminals — Part 10: Dialogue principles. ISO 9241-10, 1996.
- [15] G. Krasner. A cookbook for using the model-view-controller user interface paradigm in Smalltalk. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
- [16] K. Leung, L. Hui, S. M. Yiu, and R. Tang. Modeling web navigation by statechart. In *Proc 24th Intl Computer Software and Applications Conf (COMPSAC 2000)*. IEEE Computer Society Press, 2000.
- [17] J. Rice, A. Farquhar, P. Piernot, and T. Gruber. Using the web instead of a window system. In *Proc ACM Conf Human Factors in Computing Systems (CHI '96)*, 1996.
- [18] K.-D. Schewe and B. Thalheim. Modeling interaction and media objects. *Proc 5th Intl Conf Applications of Natural Language to Information Systems, LNCS*, 1959:313–324, 2001.
- [19] D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Comm ACM*, 38(8):45–46, Aug 1995.
- [20] M. Winckler, E. Barboni, C. Farenc, and P. Palanque. SW-CEditor: A model-based tool for interactive modelling of web navigation. In *Proc 5th Intl Conf Computer-Aided Design of User Interfaces (CADUI 2004)*, pages 55–66. Kluwer Academics, 2004.
- [21] T. Ziegert, M. Lauff, and L. Heuser. Device independent web applications — the author once – display everywhere approach. *Proc 4th Intl Conf Web Engineering (ICWE 2004)*, LNCS, 3140:244–255, 2004.