Journal of Web Engineering, Vol. 8, No. 1 (2009) 048–070

FINE-GRAINED SPECIFICATION AND CONTROL OF DATA FLOWS IN WEB-BASED USER INTERFACES[a]

MATTHIAS BOOK VOLKER GRUHN
Applied Telematics/e-Business Group[b], Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany
{*book, gruhn*}*@ebus.informatik.uni-leipzig.de*

Received November 18, 2007
Revised August 26, 2008

When building process-intensive web applications, developers typically spend considerable effort on the exchange of specific data entities between specific web pages and operations under specific conditions, as called for by business requirements. Since the WWW infrastructure provides only very coarse data exchange mechanisms, we introduce a notation for the design of fine-grained conditional data flows between user interface components. These specifications can be interpreted by a data flow controller that automatically provides the data entities to the specified receivers at run-time, relieving developers of the need to implement user interface data flows manually.

Keywords: Dialog Flow Notation, data flow specification, data flow control

1 Introduction

Web-based user interfaces have become popular front-ends for information systems that require convenient access at any time from anywhere [2]. Especially in business-to-business and intranet applications that are designed to support elaborate business processes, these user interfaces can turn out to be quite complex. Their complexity is typically twofold: Most obviously to the user, they have intricate dialog structures that include nested dialog sequences, wizards, context-sensitive links and other navigation patterns. More transparent for the user, but all the more palpable for the developer, are the complex data flows between the interface and the business logic. No matter if these data flows mirror major business process features or serve minor technical purposes, the developer must ensure that the right data is available for the right component at the right time, all the while keeping an eye out for security issues, validity concerns, performance considerations and persistence strategies.

Despite some progress over the past years, the WWW infrastructure itself still provides only rudimentary data flow support: Originally, the only available data flow mechanism was the transmission of parameter strings from clients to servers in HTTP **requests**. Soon, web browsers became capable of receiving short **cookie** strings from a server and sending them back to the same server with every subsequent request. This in turn enabled servers to

[a] A preliminary version of these concepts was presented in [1]. In addition, this article presents a complete introduction to all notation constructs, and evaluates their impact on web applications' development effort.
[b] The Applied Telematics/e-Business Group is endowed by Deutsche Telekom AG.

unambiguously associate multiple requests with the same client, and keep individual state information for all clients in **sessions**. Since sessions usually expire after some time without user activity, **persistence** of selected data is typically ensured by integration of databases or other storage technologies in the back-end.

These basic data flow mechanisms are technically sufficient to build any web-based application: Requests provide a channel for data flows from client to server, sessions provide a scope for data flows among server components, and if necessary, cookies can provide an additional channel for client-server data exchange, client-side state-keeping, and even simple client-side persistence. However, these mechanisms are only convenient for a small subset of conceivable data flows: Requests are ideal for sending data from a web page to those business operations responsible for building the server's response; and sessions are ideal for making data accessible throughout the application. These alternatives mark two ends of a spectrum – in our experience, however, most data flows described by business processes lie somewhere in the middle: Often, data generated in a certain process step is intended only for a clearly defined, but not necessarily immediately following set of pages and/or operations. Thus, the respective data flows must reach beyond a single request-response cycle, but do not require session-wide data publication.

When mapping such process requirements to the technical level, developers currently need to choose the lesser of two evils: Passing data along a chain of requests toward its ultimate destination violates the principle of encapsulation, as the intermediate pages and operations need to handle data that they are not actually responsible for – an unclean and error-prone solution that requires high implementation and maintenance effort. Alternatively, storing data in the session for use at a later time relieves the intermediate steps from a lot of hassle, but bears the danger of memory leaks if the data is not removed from the session when it is no longer required. In addition, both approaches pose inherent security risks as data is exposed to pages or operations that do not need to know about it (and in the first case, even repeatedly sent over the network). Even if other application components are considered "friendly", this unnecessary exposure multiplies the possible points of failure or attack.

Since neither the request nor the session scope provides satisfactory data flow support, we propose a supplemental method for realizing data flows that match the process requirements of application domains more closely, while at the same time reducing the required implementation effort. In this paper, we will first show how arbitrary point-to-point data flows can be specified as an extension to a graphical dialog flow notation (Sect. 2). We then show how this specification can be interpreted at run-time by a data flow controller that provides just the specified data to each page and operation (Sect. 3). After an evaluation of the development effort that may be saved using this approach (Sect. 4), we conclude with a discussion of related work (Sect. 5) and an overview of further research opportunities (Sect. 6).

2 Data Flow Specification

In the introduction, we discussed data flows related to the user interface layer. Obviously, data also flows along the control structures within the application logic, may be exchanged with third-party, legacy or back-end systems, and can be stored in and retrieved from persistent memory. However, since these data flows are the responsibility of the application and persistence logic, established specification methods (e.g. UML data flow diagrams, sequence

diagrams etc.) and implementation techniques (e.g. SOAP, EJB etc.) can be used for these layers. Our work instead focuses on data flows between the presentation and application logic, where suitable implementation techniques (apart from the coarse request and session scopes) and specification methods have not yet been widely established.

To specify user interface data flows on a more fine-grained level than the one provided by the request or session scopes, we first need to define the concept of a data flow more clearly. Given a data source A, a data sink B and a data entity d, we define that the data flow of d from A to B is the provision to B of the d available to A (i.e. B gets to know the d that is known to A). As we will soon see, it is helpful to make data flows conditional, i.e. to execute them only if a certain constraint is fulfilled.

In designing a notation that maps this abstract definition onto a technical level, we need to answer a number of questions:

- What are concrete data sources and data sinks?
- Which constraints determine if a data flow is executed?
- How are sources and sinks related?
- What are concrete data entities?
- How is the "provision" of data entities realized?

The first three questions relate to the specification of data flows. We will answer these in the following sections 2.1–2.4 by showing how an existing notation for specifying dialog flows can be extended with new constructs for specifying data flows. The last two questions relate to the implementation of data flows. We will answer those in Sect. 3.1 when we are considering fundamental decisions that need to be made before detailing the design of a framework that will handle the specified data flows at run-time (Sect. 3.2).

2.1 Data sources and sinks

To address our first question, the data sources and data sinks used to model data flows should be entities that represent the structure of web-based user interfaces. In our past work, we have found it natural to distinguish between web pages and application logic operations in order to model the navigation structure of web applications: In the Dialog Flow Notation (DFN) [3], web pages are symbolized as dog-eared sheets (the so-called **masks**), while application logic operations (**actions**) are symbolized by circles. To specify all possible navigation paths, these elements are connected by arrows symbolizing **dialog events** generated by masks or actions when the user submits a request, or a business operation produces a result. In the dialog graphs specified this way, masks and actions do not have to alternate since it is conceivable for a mask to trigger a succession of several server-side operations, and also conceivable for a mask to link directly to another mask without the need for any intermediary business operations. The dialog graphs are encapsulated in **dialog modules** that can call each other at run-time to form hierarchically nested dialog structures.

In our current work on data flows in web applications, we found that the DFN provides an ideal basis for specifying the data flows within a web-based user interface: The masks, actions and modules can serve as data sources and sinks, while the dialog events can be interpreted

as constraints that determine when a data flow is executed. In the following sections, we will show how the original DFN can be extended to specify different kinds of data flows. Our aim here is to extend the notation in such a way that the data flows are intuitively readable and conceptually integrated with the DFN semantics, but do not add unnecessary optical clutter to the diagrams.

2.2 Data flow types

Depending on the types of the data sources and sinks, and their proximity within the topology of the dialog graph, we introduce several kinds of data flows:

2.2.1 Parallel data flows

The simplest type of data flow is running in parallel with the dialog flow, i.e. a data flow from the generator of a dialog event to the receiver of the same event. It is specified in the DFN by adding the label of the data entity in square brackets to the event's name (if the data flow shall comprise several data entities, we separate them with commas). For example, in the *login* module in Fig. 1, the *name* and *passwd* data entities entered by the user on the *login* mask shall only be provided to the *check name, passwd* action upon generation of the *submit* event. At first sight, this data flow may look equivalent to the request scope, but it is actually more finely grained: While the request scope always encompasses all intermediate actions up to the next mask, the parallel data flow is restricted to the current dialog event's receiver only. Therefore, the *passwd* data in the example is only provided to the succeeding action, but not accessible beyond it (as it would be in the request scope).

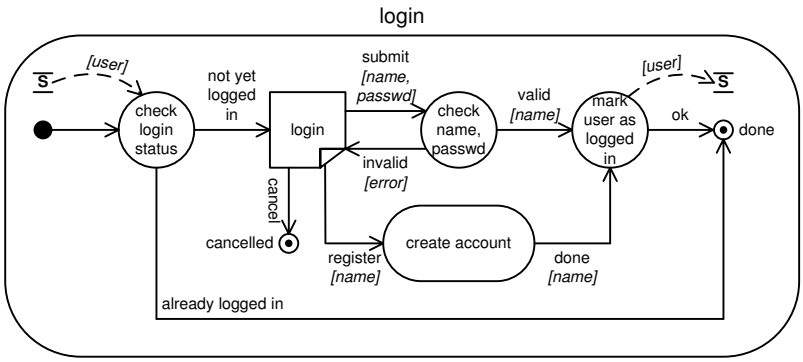

Fig. 1. *Login* module performing user authentication.

Request scopes and parallel data flows also differ in their starting points: While a request can only be generated by a mask, a dialog event with associated parallel dialog flow can also be generated by actions or modules. For example, if the login credential check fails, the *error* data is provided by the *check name, passwd* action to the *login* mask with the *invalid* event.

Finally, parallel data flows can be used more flexibly than request scopes since developers can specify which data is provided with which events: Using only the request scope, the developer may not be able to prevent that the credentials entered by the user on the *login* mask are provided to subsequent elements both for the *submit* and (unnecessarily) the *cancel*

event. Using parallel data flows, however, the developer can specify that the *name* and *passwd* data is provided to the subsequent action with the *submit* event, just the *name* data is provided to the *create account* module with the *register* event, but no data is provided with the *cancel* event.[a]

2.2.2 Divergent data flows

As we have just seen, parallel data flows enable developers to specify which data entities shall be provided to the receivers of which events. Often, however, the data generated by one element should not be provided to any of its immediate successors, but rather to some more distant element in the dialog graph that is responsible for the actual data processing.

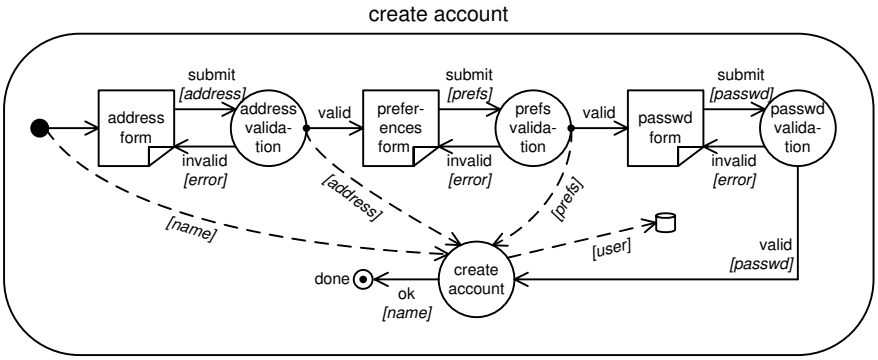

Fig. 2. *Create account* module employing a "wizard" navigation pattern for gathering user input.

As an example, consider the "wizard" navigation pattern of the *create account* module in Fig. 2: Here, we prompt the user for his address, preferences and password, validate each of the inputs, and finally create an account from all the collected data. In order not to burden the various forms and validation actions with forwarding the data of their predecessors, we would like to provide the data gathered in each step directly to the final processing action. For the validation actions, the outgoing dialog and data flows therefore diverge: For example, if the *address validation* action produces a *valid* event, the dialog flow shall continue to the next wizard step, i.e. the *preferences form*, but the *address* data shall be provided directly to the *create account* action, bypassing the other steps of the wizard. To specify that the data flow diverges from the dialog flow in this case, we draw a dashed line to the element that the data shall be provided to, and annotate it with the data entity's label in square brackets.

Just like a parallel data flow is always tied to a dialog event, a divergent data flow is also always associated with a particular dialog event and only executed if that event is traversed. To specify this constraint in the DFN, the data flow arrow must always begin in the same spot (marked by a black dot) as the event arrow that it shall be associated with. To maintain consistency with parallel data flows, which are always implicitly associated with an event, the DFN does not allow the specification of divergent data flows without an associated event.

[a]Note that these restrictions cannot prevent the respective data from being transmitted across the network in the request, as this would require client-side logic. However, the data flow controller will ensure that any submitted data is only available to the specified receivers on the server side.

2.2.3 Inter-module data flows

Of course, data may not only flow among the dialog elements within a module, but also between modules. Therefore, when a module is called, it should be able to accept data provided to it, and when a module terminates, it should be able to provide data to subsequent dialog elements. To express this behavior in the DFN, compatible data flows must be specified both in the exterior dialog graph that a module is embedded in, and in the interior dialog graph of the module itself: In the exterior dialog graph, modules can simply serve as sources and sinks of parallel and divergent dialog flows just like masks and actions. In Fig. 1, for example, the *create account* sub-module is provided *name* data with the incoming *register* event and provides *name* data with its outgoing *done* event.

To specify which data entities a module can accept and provide, we employ the initial and terminal anchors of its interior dialog graph as data flow interfaces: Parallel and divergent data flows can originate from the **initial anchor** (the black disk marking the starting point of the dialog graph traversal) to specify to which elements the incoming data shall be provided. Analogously, parallel and divergent data flows can lead to the module's **terminal anchors** (the circled small black dots marking the end of the dialog graph's traversal). They specify which data entities the module will provide to its exterior dialog graph, and who provides those data entities internally. In the definition of the *create account* module in Fig. 2, for example, the incoming *name* data flows directly from the initial anchor to the *create account* action, which will process it later together with the other data collected by the wizard. The *name* data will then flow from the *create account* action to the *done* terminal anchor, so it can be provided to the module's successor in the exterior dialog graph upon its termination.

In this example, the names of of data entities in the exterior and interior dialog graph match. However, this may not always be the case: In the interest of encapsulation, it should be possible to provide data to a module even if that data entity is known by a different name inside the module. Similarly, it should be possible to work with data produced by a module independently of the entity's label inside the module. In order to facilitate this decoupling of name spaces, the extension to the DFN allows the renaming of data entities provided to or by a module in its exterior dialog graph. We can distinguish three cases:

- For a data flow leading to a module, the notation $a \rightarrow b$ indicates that the data entity known under the label a in the module's exterior graph should be available under the label b in its interior graph.

- For a data flow produced by a module, the notation $x \rightarrow y$ indicates that the data entity known under the label x in the module's interior graph should be available under the label y in its exterior graph.

- For a data flow between two modules embedded into the same super-module, the notation $s \rightarrow t$ indicates that the data entity known under the label s in the first module's interior graph should be available under the label t in the next module's interior dialog graph (this is a special case insofar as the exchanged data never becomes known to a module's exterior dialog graph, but is channeled directly from one interior dialog graph into the next).

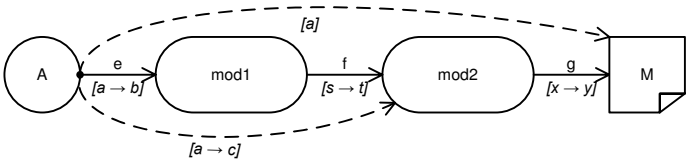

Fig. 3. Renaming inter-module data flows.

Figure 3 illustrates the different options: The data entity a produced by action A is provided to module *mod1* under the label b, to module *mod2* under the label c and to mask M under the same label a. Note that data flows which do not involve any modules cannot be renamed, as atoms in the same module will usually be semantically related and thus do not require different names for the same data entity.

Through these incoming and outgoing interfaces, data can be flexibly passed from several sources outside a module to several sinks inside a module and vice versa, while keeping the name spaces of the modules independent of each other. Developers need to ensure that the data flows specified in the interior and exterior dialog graphs match, as unmatched entities will otherwise be unavailable to their sinks.

2.2.4 Wildcard data flows

In many web applications, certain pivotal pages like the home page, or certain features like a login/logout mechanism, should be available from virtually anywhere within the dialog flow. To prevent the redundancy and combinatorial explosion of events that would ensue if we modeled all the required events connecting these pages explicitly, the DFN provides two types of "wildcard events".

By drawing a so-called **compound event** from a module definition's contour to an element inside the module, developers can indicate that the respective event may be generated by any element inside that module to reach the designated receiver. If the visibility of this event shall not only encompass the current module, but also extend to all its direct and indirect sub-modules, developers can specify a so-called **common event** with such an expanded scope by letting the event point to a module outside the contour.[b]

As an example, consider the *last minute* booking module of a travel portal depicted in in Fig. 4. In order not to restrict users to a certain booking order, the *Flight Offers*, *Hotel Offers* and *Rental Car Offers* masks are all reachable from each other and from the *Itinerary* summary mask through the compound events *flights*, *hotels* and *cars*, respectively. Each of the masks accepts a *dest* parameter indicating the desired destination, and provides individual booking data to the *booking* module. While the offer masks are available throughout the *last minute* module, they cannot be reached from inside the *booking* module. In contrast, the *context help* module is reachable through the common event *help*, indicating that it can be called from within both the *last minute* module and its nested *booking* module. As the parallel data flow *context* indicates, any dialog element calling this module from anywhere will have

[b] The DFN prohibits that common events lead to masks or actions, as these cannot be sensibly "nested" into sub-compounds. In [3], we discuss in more detail how to return from a module even if no receivers could be specified for its terminal events at design-time.

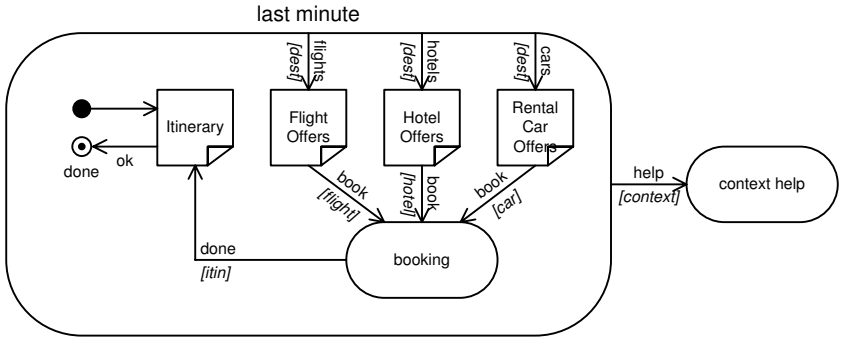

Fig. 4. *Last minute* module for booking trips in a travel portal.

to provide context information in order to show the relevant help message.

From the perspective of data flows, the interesting question is how to handle data flows associated with compound and common events and their receivers. The challenge here lies in the fact that at design-time, the developer does not know which elements will actually generate the wildcard events (and their associated data flows) at runtime.

Since any element of a module's dialog flow can be a receiver of a compound event, we place no restrictions on these element's capabilities to accept and provide data. An element receiving a compound event may thus accept and produce parallel and divergent data flows just like any other element.

Modules that are receivers of common events (so called "common modules") are a more special case since they are not only part of the dialog graph of the module that they are tied to, but are implicitly also part of the dialog graphs of all its sub-modules. On the level of the dialog flow, this means that developers cannot explicitly specify the receivers of any events produced by the common module. On the level of the data flow, it follows that we cannot specify receivers of data flows produced by common modules, as we do not know at design-time which contexts they may be embedded in at runtime. For a similar reason, common modules cannot accept divergent data flows: While it would be possible to draw a divergent data flow to a common module in the module that it is defined in, it would not be possible to specify such data flows in the sub-modules that it is also visible in. For consistency, the DFN therefore allows only parallel data flows to lead to common modules.

Of course, this begs the question how parallel data flows (i.e. data flows where the data source and sink are identical to the event's generator and receiver) should be interpreted for compound and common events, since the generators of compound and common events are by definition unknown at design time. In analogy to data flows for explicit events, we understand parallel data flows of wildcard events as a requirement that any element generating such an event must also provide the specified data to the receiving element.

2.3 Data flows in abort mode

As described in [3], there are situations in which it may be necessary to abort the traversal of a module's dialog graph even before one of its terminal anchors has been reached. To facilitate a controlled termination in such a situation, the DFN provides the so-called "abort

graph" for the specification of a simple dialog graph that may comprise user confirmation and cleanup operations. This dialog graph (which must be isolated from the module's regular dialog graph) begins at a cross-shaped **abort anchor** and terminates at a special "canceled" anchor (symbolized by a circled cross). As an example, Fig. 5 shows an extended version of the *create account* module we presented earlier. When the traversal of this wizard is aborted, the abort graph specified in the module's upper half is triggered by the dialog control logic. The *store input* action stores all data entered up to now in the user's session, so when the user picks up the registration process again at a later time, the input fields can be populated with the previously entered information.

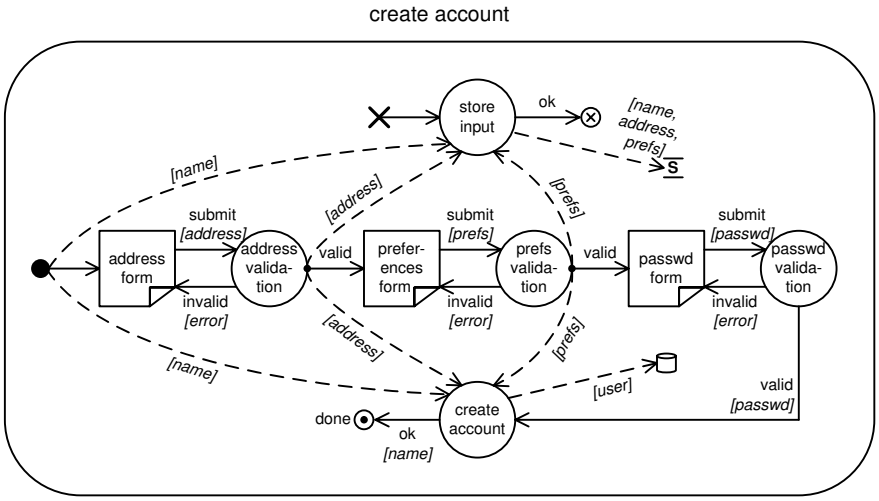

Fig. 5. *Create account* module storing incomplete information in session, if aborted prematurely.

When specifying data flows, developers can treat a module's abort graph quite similarly to a regular graph, with two exceptions: Firstly, since the abort graph is never called explicitly by the developer, but invoked by the dialog control logic when necessary, there are no explicit events leading to the abort graph. Consequently, it is impossible to provide data to the abort anchor, and thus also nonsensical to specify data flows originating from it. Secondly, an aborted module does not generate a terminal event that has a definite receiver in its exterior dialog graph, but the dialog control logic determines the further course of action. For this reason, it is nonsensical to provide data to the "canceled" anchor of an abort graph.

While the abort graph needs to be isolated from the regular dialog graph in terms of dialog events, it is possible to pass data between elements of either graph through divergent data flows. In the example in Fig. 5, the *store input* action in the abort graphs receives all data entered so far from the masks in the regular graph in the lower half of the module.

2.4 Shared scope access

Parallel and divergent data flows enable developers to define the propagation of data entities on a very fine-grained level. However, their specification may become cumbersome if the same data entities shall be provided to many elements, as a profusion of explicit data flow arrows

would be required. For this reason, the DFN provides constructs to symbolize data exchange through various shared scopes.

2.4.1 Module scope

Often, the same data entities should be provided to virtually all elements within the same module. As we mentioned earlier, storing such data entities in the session scope is not an optimal solution for this problem since the developer would be responsible for removing them from the session before the module is terminated. Instead, it would be preferable if all elements within a module had a shared scope that is cleared out automatically once the module terminates. We have extended the Dialog Control Framework (DCF) with the **module scope** for this purpose: It is comparable to the session scope in that it is associated with the current user, but exists only during the traversal of the current module. An empty module scope is instantiated whenever a module is entered, and discarded again once a terminal anchor has been reached (we will describe the implementation details in Sect. 3).

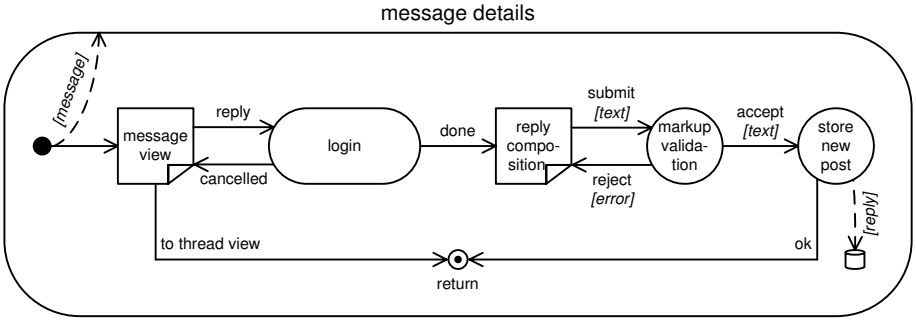

Fig. 6. *Message details* module for viewing and replying to messages in a discussion forum.

To specify in the DFN that a data entity should be provided through the module scope, developers can simply draw a divergent data flow with the respective entity label in square brackets from any dialog event to the module's contour. In the *message details* module in Fig. 6, for example, the incoming *message* data is provided to all other elements through the module scope right away.

Of course, data flows to the contour can also originate from any other source, indicating that the respective data will be available throughout the module as soon as the associated event is traversed. Since all data in the module scope is implicitly available to all dialog elements, no explicit notation construct is required to indicate that an element accesses data from the module scope. In the *message details* module, for example, all masks and actions can access the original *message* data in the module scope to display it, quote it in the text of the reply, and reference it with the new post. However, data in the module scope is not automatically made available to sub-modules, as this might constitute an undesired expansion of the data's visibility (especially in the presence of common modules). To leave developers in control of how broadly they would like to provide data, data flows to modules must always be specified explicitly. In this way, the DFN supports the basic programming pattern of separation of concerns.

In the DCF implementation, module scopes are stacked to reflect the nested module call structure: When a module calls a sub-module, a fresh module scope for the sub-module is pushed onto the stack, rendering the calling module's scope temporarily inaccessible. When the sub-module terminates, its module scope is removed from the stack, and the calling module's scope becomes available again.

2.4.2 Session, application and cookie scope; storage access

The mechanisms described so far are helpful in many situations where the data scopes provided by the application server are too coarse for the data flow requirements at hand. However, there are obviously also a number of situations where those more generous scopes are the perfect choice – information about the user's login status, for example, should be available throughout the application and is therefore ideally stored in the session scope. Other data may best be globally provided through the application or cookie scope, and business objects often need to be stored in or retrieved from persistence storage by the application logic.

The data flows for these scenarios usually concern only the application (and possibly persistence) layer, so the dialog control logic that couples the presentation and application layer should not be involved in them. Consequently, the DCF does not handle data flows through these larger scopes, but lets the business logic interact directly with the application server or persistence layer API. This restriction of responsibilities allows for a clean separation of concerns on an architectural level, and allows developers to choose whichever scope and persistence framework is suited best to their needs, instead of being bound to the DCF's data flow mechanisms throughout the application.

Due to the framework's limitation to user interface management, the DFN technically would not need to provide additional constructs related to the larger scopes. Intuitively, however, developers will expect to be able to specify not only fine-grained data flows through the user interface, but also data shared through the more coarse scopes.

The DFN solves this dilemma by providing a compromise – developers cannot specify the complete application logic's internal data flows within the dialog flow, but the DFN enables them to specify where the dialog flow interfaces with the larger scopes: Data flow arrows leading to or from the letters **S**, **A** or **C** enclosed by horizontal bars indicate that the respective data entities are stored in or retrieved from the session, cookie or application scope, respectively. For example, in Fig. 1, the *check login status* action retrieves *user* data from the session scope, and the *mark user as logged in* action stores *user* data in the session scope. In addition, data flow arrows leading to or from a "can" symbol indicate that the respective data entity is stored in or retrieved from persistent storage – for example, the *user* data constructed by the *create account* action in Fig. 2 is sent to the back-end for storage. Note that in contrast to divergent data flows, the data flows leading to those larger scopes cannot share their starting point with an event, since the respective data handling operations are beyond the control of the dialog controller that is only aware of dialog events.

Since the DCF does not provide mechanisms for handling these scopes, the corresponding notation constructs have only illustrative character – the actual data handling will have to be implemented manually by the developer. In contrast, the module scope and parallel, divergent and inter-module data flows are executable specifications: The developer only needs to specify the desired behavior, but does not need to implement it since the framework handles the

respective data flows automatically.

2.5 Summary

In Sect. 1, we motivated the need for an explicit specification of data flows in web applications, with the aim of letting a framework control them at run-time, so developers do not need to bother with these technicalities, but can focus on the implementation of the business logic. In the preceding subsections, we have introduced a notation extension providing various constructs for this purpose.

One might wonder whether a domain-specific language for data flow specification in web applications is necessary, since various notations for specifying data flows in software already exist. However, we found these approaches insufficient for expressing the particular issues that developers of complex web applications face (and typically solve through tedious manual implementation). As an example, consider the comparison of the DFN diagram and UML activity diagram in Fig. 7 that both describe the same simple dialog and data flows.

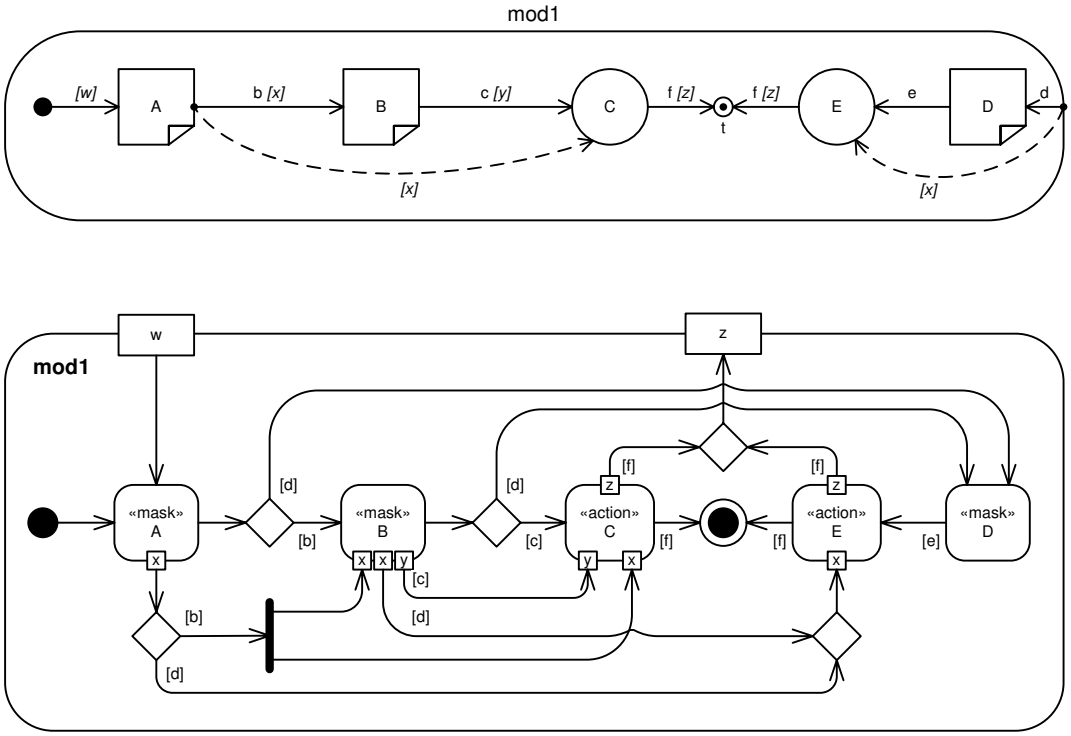

Fig. 7. Comparison of a DFN diagram (top) and equivalent UML activity diagram (bottom).

While the intuitiveness of notations may be hard to measure directly, even this simple example already shows that the DFN's visual complexity is significantly lighter (in terms of the number of nodes, edges and intersections). A more important limitation of UML activity diagrams is that some constructs (namely the module scope, and data flows associated with compound events) could only be expressed if one accepts combinatorial explosion of edges;

and data flows associated with common events cannot be expressed at all (since their availability may depend on which super-modules the current module is invoked from at run-time; an aspect that cannot be modeled in activity diagrams). Some of these control flow limitations could theoretically be ameliorated by the use of UML state machine diagrams, which however do not allow the specification of data flows. We believe these limitations warrant the introduction of a domain-specific language tailored to dialog and data flow specification for web applications.

Using the extended DFN's new finely-grained data flows, the DCF's module scope and the traditionally available scopes, developers have a full spectrum of data flow mechanisms at their disposal, as Fig. 8 illustrates: While the application and session scopes are suitable for publishing application-global and user-global data, parallel and divergent data flows are ideally suited for data propagation between arbitrary dialog elements. Since multiple parallel and divergent data flows can be associated with any dialog event, developers can control which data is propagated where under which conditions very flexibly.

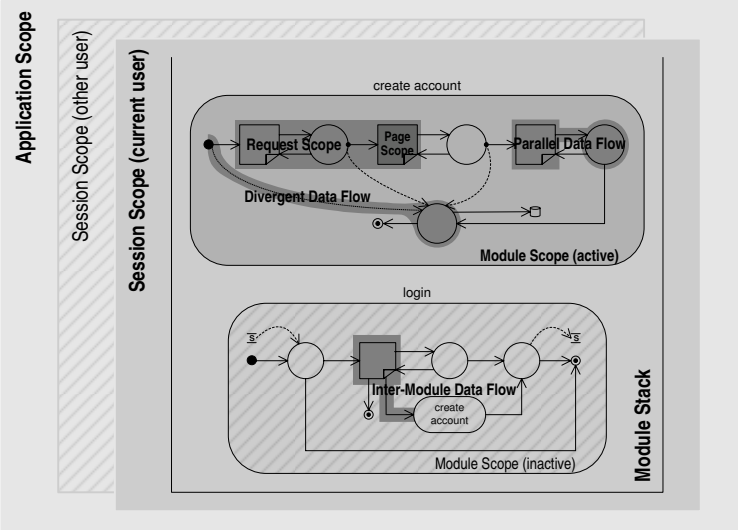

Fig. 8. Scopes and data flows supported by Dialog Flow Notation and Dialog Control Framework.

The page scope remains an important tool for data exchange between the components involved in constructing a response (e.g. a JavaServer Page (JSP) and its tag libraries). The request scope, however, has been rendered virtually redundant since its visibility is determined by purely technical criteria that only seldom map exactly to business requirements for data propagation. This allows us to retrofit it for conveniently accessing the new data flow mechanisms, as we will show in the following section.

3 Data Flow Controller

Having introduced the new data flow constructs of the extended Dialog Flow Notation, we will now show how their semantics are supported by the Dialog Control Framework (DCF) that controls applications according to these specifications. Revisiting our initial questions from

Sect. 2, we first need to address the technical representation of data entities, and discuss how the provision of those entities from sources to sinks is interpreted (Sect. 3.1). After this, we will give a brief overview of the technical implementation of the data flow control mechanism that we integrated with the DCF (Sect. 3.2).

3.1 Data entity representation and provision

In the previous sections, we identified data entities only by their label (e.g. "*user* data"). In concrete applications, a data entity can be any object – in Java-based applications, it will typically be an instance of a JavaBean holding various attributes, which is stored in or retrieved from a data scope using its label as a look-up key. In Java Enterprise Edition-compatible application servers, for example, `HttpSession` instances provide the `Object getAttribute(String name)` and `void setAttribute(String name, Object value)` methods for this purpose. Just like the DFN relies on existing notations to specify the layout of dialog masks (e.g. visual design sketches or XForms) or the control flow within actions (e.g. UML activity or state diagrams), it does not provide its own constructs for the specification of data entities' internal structure. Instead, we recommend using existing notations such as entity-relationship or UML class diagrams for this purpose.

Since we assume that the data entities in our data flows are objects (i.e. reference types), we have two alternatives for interpreting the flow of data from a source to a sink: "providing a data entity" could mean

- forwarding the data entity itself from the source to the sink, or
- extending the data entity's scope so it is not only available to the source, but also to the sink.

While these alternatives may at first sight look like equivalent implementation variants, they exhibit different behavior if a source provides the same data entity to several sinks, as illustrated in Fig. 9. In a situation like this, the first approach (forwarding the data entity) intuitively implies that both B and C receive identical copies of d from A, and that any changes B makes to d will not affect C's copy of d. In contrast, the second approach (extending the entity's scope) implies that B and C can now both access the same instance of d that is already known to A, so any changes that B makes to d will also affect the d available to C. In short, the first approach requires a copy-by-value mechanism, while the second can be implemented as copy-by-reference.

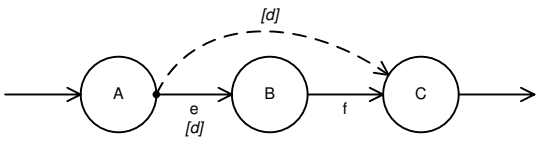

Fig. 9. Data flow with different copy-by-value and copy-by-reference behavior.

The copy-by-value implementation has the advantage that side effects are avoided. However, since we are dealing with objects that may have arbitrary complexity, copying data entities is not trivial. We could simply require that all data entities are cloneable and thereby

put the responsibility on application developers, however, this is hardly a satisfactory solution for a framework that strives to make developers' jobs easier. Furthermore, cloning comes at a high performance and memory cost (especially since some clones may turn out to be unnecessary if the respective data sinks never happen to be visited in the user's subsequent traversal of the dialog graph), and some data structures may not be cleanly cloneable with justifiable effort. A restriction to flat copies or primitive types is not a realistic option either, since it would limit application developers' design freedom severely.

For these reasons, we prefer a copy-by-reference implementation of data flows. While it may involve a bit more subtle semantics, its basic concept and the situations in which side effects may occur are well known to experienced developers. This approach is much easier to implement for framework and application developers, and does not cause the performance and memory overhead of cloning.

The above considerations were confirmed by our initial prototype of a data flow control extension for the DCF: Here, clones of JavaBeans were only flat copies, so any nested references were not cloned, and the isolation between the data entities available to the dialog elements was not perfect. We have meanwhile switched to a data flow controller implementation that copies only object references instead of whole instances, and thus realizes the scope extension.

3.2 Data flow controller design

Having determined the operational semantics of data flows, we still need a way to actually provide the specified data to the respective dialog elements in a web application. At first sight, an obvious solution would be to equip every mask and action instance with a look-up table that holds references to all data entities available to that element. This individual "element scope" could then be accessed by the application and presentation logic just like the session or application scope, using `setAttribute` and `getAttribute` methods. A data flow controller would ensure that object references are copied from one element scope to another according to the data flow specifications.

While this approach seems straightforward, it proves cumbersome in practice since the element scope itself cannot be made easily available to the presentation logic: Servlets, JSPs and other web resources provide convenient mechanisms for accessing the standard scopes (e.g. through the implicit `application`, `session` and `request` objects of the Java EE Expression Language), but do not provide as convenient means for accessing custom-built scopes.

In the data flow extension to the DCF, we therefore took a slightly different approach: The data flow controller still manages look-up tables that contain the object references available to each dialog element. However, these tables are not directly accessible to the dialog elements. Rather, we project their contents into the existing request scope that is already conveniently available to all masks and actions.

As Fig. 10 shows, this can be achieved by wrapping the original HTTP request into a wrapper object (here, `AugmentedRequest`) that looks and behaves just like an HTTP request, since it passes most method calls directly to their counterparts in the original `HttpServletRequest`. The only exceptions are the `setAttribute` and `getAttribute` methods that are normally used to access the request scope, but now rerouted to the `DataFlowController`. This central controller has (via the current `Module` on top of the `ModuleStack`) access to the data flow model (a graph of `DialogElement`s linked with `DataFlow` edges), the current `ModuleScope` and the

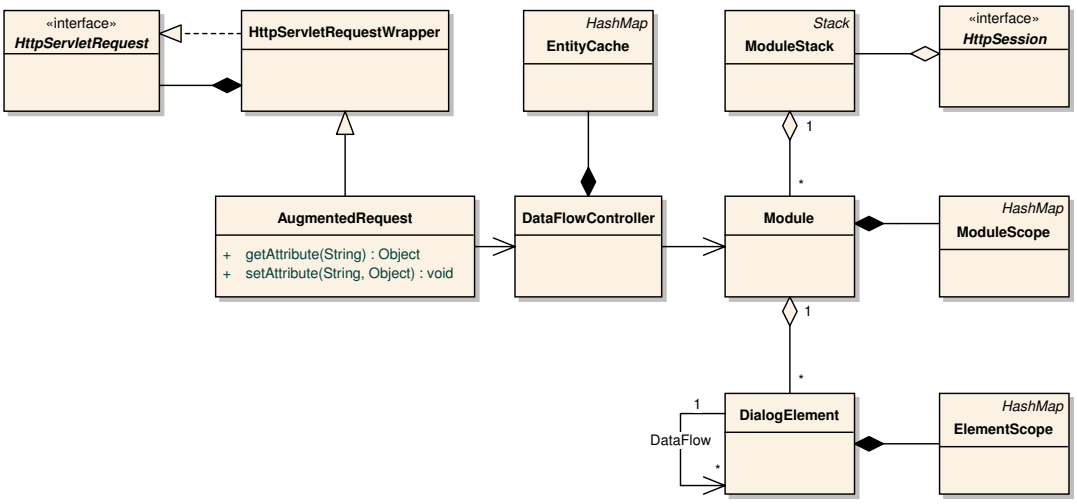

Fig. 10. Data flow control logic of the Dialog Control Framework.

current element's `ElementScope` (two `HashMap`s containing the object references associated with the data entity labels).

Whenever a mask or action calls the request's `getAttribute` method with some data entity label, the `DataFlowController` looks up the respective object reference in the current `ModuleScope` and `ElementScope` (the element scope takes precedence over the module scope in case both contain a reference with the same label), and returns the found reference.

Populating the look-up tables requires slightly more effort: Whenever a dialog element calls the request's `setAttribute` method, the `DataFlowController` stores the respective data entity in the current element's scope. It also looks up any departing data flow edges in the module's data flow model and caches the entity's object reference if it is associated with any outgoing event. Once an event is generated by the current dialog element, the associated data references from the `EntityCache` are stored in the element scopes of the respective data flow receivers, and the cache is cleared.[c] This way, the data flow controller ensures that every element can access through the request context all data entities available to it according to the specification.

Besides giving the presentation and application logic convenient access the element and module scopes, projecting the element scope into the request scope has the additional benefit that the mechanism is transparent to other web application frameworks: In our Java EE-based reference implementation of the above design, we have integrated the dialog and data flow control logic with JavaServer Faces (JSF) to make use of its UI component model and validation logic. Since JSF pages exchange data through the request scope, they can work with the new module scope and data flows seamlessly.

[c] Our actual data flow controller implementation is a bit more complex than the diagram and this brief description suggest – among other things, it also moves incoming request parameters into request attributes, and ensures that data references are also copied to other scopes if they have not been set by calls to `setAttribute`, but received through data flows from other elements.

4 Evaluation

In order to get an impression of the impact of our approach to data flow specification and control on software development practice, we consider another scenario from the travel portal example: To collect statistics on the users' favorite destinations, activities etc., we collect users' votes in a series of masks, and update the statistics in a final *tally votes* action. As Fig. 11 shows, we can use three divergent and one parallel data flows to gather the various entries from the masks and provide them all to the final action.

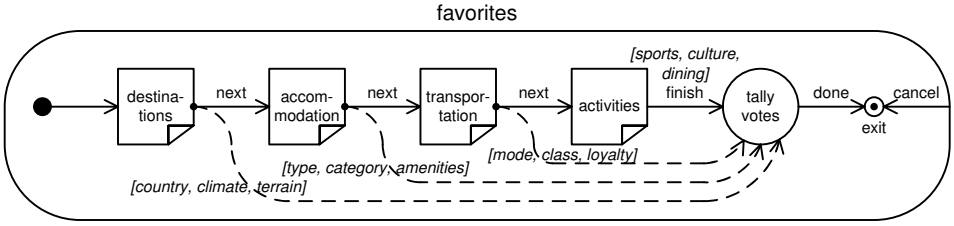

Fig. 11. *Favorites* module for accumulating statistics on users' travel preferences.

To compare this approach against traditional manual data provision techniques, we will evaluate a number of criteria: By identifying function points (FP), we can obtain a rough indication of the required specification and implementation effort. By considering coupling and cohesion, we can gauge the complexity of the implementation, and the resulting testing and maintenance effort. As we will see, these criteria can be examined along multiple dimensions within and in between application layers.

4.1 Implementation using data flow controller

When realizing the voting feature using the DCF's data flow control mechanisms, we can identify the following function points for a sequence of p pages that each prompt for e data entities:

- **Dialog flow specification:** While we do not need to implement transitions between dialog elements when using the DCF, we do have to specify them. In the Dialog Flow Specification Language (DFSL) that can be automatically generated by our graphical Dialog Flow Editor for parsing by the framework, these transitions are expressed as XML elements such as

```
<ex-mask name="accommodation">
  <on-event name="next">
    <call-mask>transportation</call-mask>
    [...]
  </on-event>
</ex-mask>
```

  If we disregard the events leading from/to the initial and terminal anchor (since they are universally necessary and not specific to the issue of data flow control) we are left with one transition going out from each page, i.e. p FP.

- **Data flow specification:** We need to specify each provision of a data entity from one dialog element to another. In DFSL documents, these are specified in place of the above ellipsis in the XML elements such as

```
<prov-data name="category">
  <to-action>tally votes</to-action>
</prov-data>
```

  Since we must specify such expressions for each entity on each page, we have $p \times e$ FP.

- **Entity generation:** To provide the desired data to the application, the dialog masks must contain input fields for all data entities that we are interested in, such as

```
<input type=radio name="category" value="Budget"/>
<input type=radio name="category" value="Standard"/>
<input type=radio name="category" value="Luxury"/>
```

  Since a group of radio buttons yields one data entity, we consider it as one function point. For a whole dialog sequence, we thus also count $p \times e$ FP.

- **Entity retrieval:** Before we can process the business data, we need to retrieve each entity from the scope it has been held in. The DCF's data flow control logic will provide the data in the request scope, where it can be accessed by Java statements like

```
String category = request.getParameter("category");
```

  Again, we count $p \times e$ FP for these statements. (We do not include the actual tallying logic, since it is identical for all techniques and thus not relevant for our comparison).

- For the *favorites* module with $p = 4$ and $e = 3$, we arrive at a total number of 40 FP in this way.

Looking at the level of coupling, we find that the implementations of all masks are independent of each other, and that the presentation and business layers are independent of any technicalities of the dialog and data flow layer. Coupling across the horizontal (inter-mask) and vertical (inter-layer) dimensions can thus be considered very loose. Horizontal and vertical cohesion within the components is quite high on the other hand, since each dialog element only comprises code that is directly related to its business aim and layer scope.

4.2 Implementation using request parameter passing

Alternatively, if we did not have the data flow controller available, we would need to implement the data propagation to the tallying action manually. In that case, we could either use the request or the session scope to hold the data until we have reached the final action. To compare the different approaches, we will first consider the strategy of passing the entered data from page to page with each request, using the hidden input fields provided by HTML. Using this approach, we would have to realize the following function points:

- **Dialog flow specification:** The dialog flow must be specified just the same as in the previous scenario, so we also count p FP.

- **Entity generation and forwarding:** Just as in the previous scenario, we need input fields for all data entities we are interested in, which we will count as one FP each. In addition, however, we also need to retrieve previously submitted data entries from the request, and carry them over into the next request by providing them in hidden input fields again – in the *accommodation* mask, e.g., we would need the following three hidden fields populated through JSP expressions with the destination data, in addition to the fields prompting for the accommodation data itself:

  ```
  <input type=hidden name="country" value="${param.country}"/>
  <input type=hidden name="climate" value="${param.climate}"/>
  <input type=hidden name="terrain" value="${param.terrain}"/>
  ```

  In the *transportation* mask, we would then need six hidden fields for the destination and accommodation data, plus three entry fields for the transportation data, etc. The number of function points can thus be calculated by the Gaussian formula $p \times (p+1) \div 2 \times e$.

- **Entity retrieval:** Retrieving all entities from the request scope for ultimate processing again requires $p \times e$ lookups (i.e. FP) in the request scope.

- For the *favorites* module, we calculate a total of 46 FP with this approach.

In this implementation, horizontal coupling between all masks is obviously very tight, since each mask also contains hidden fields for any data acquired in any previous mask. Vertical coupling between the presentation and dialog/data flow layer is also tighter, since each mask must actively forward data to its successor. Consequentially, cohesion is low in both dimensions.

4.3 Implementation using session scope

As an alternative to passing entered data along the sequence of requests, we can also store it in the session scope once collected, and retrieve it only when ultimately needed. While this initially seems like an elegant solution, it requires an intermediate step after each page in order to copy the entered data from the (transient) request scope to the more permanent session scope. This can be accomplished by interleaving auxiliary actions with the masks in the dialog flow, as shown in Fig. 12.

An implementation of this approach would require the following function points:

- **Dialog flow specification:** In the dialog flow, we need to specify the double number of transitions for each mask shown to the user, since we first need to invoke the data preservation action (e.g. *hold accommodation*) before we can invoke the subsequent mask. The only exception is the last mask, which can provide its parameters directly to the tallying action through the request context. We thus need to specify $p \times 2 - 1$ transitions (i.e. FP).

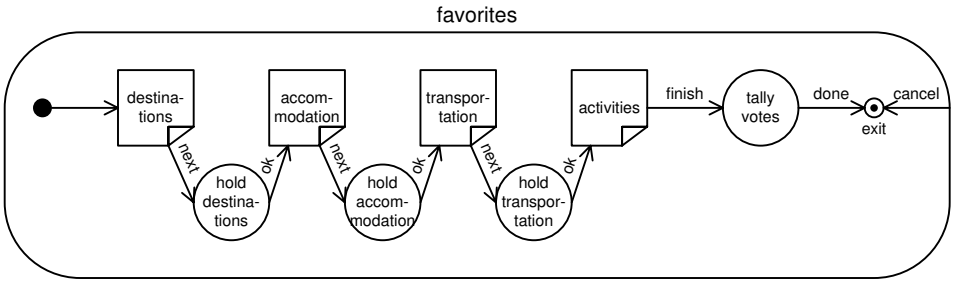

Fig. 12. *Favorites* module with actions for preserving user request parameters in session attributes.

- **Entity generation:** To prompt the user for data, we need the same number of input fields as in the first approach, i.e. $p \times e$ FP.
- **Entity preservation:** After submission of every mask, an intermediate action must retrieve the data from the request scope and preserve it in the session scope through Java statements such as

  ```
  session.setAttribute("type", request.getParameter("type"));
  session.setAttribute("category", request.getParameter("category"));
  session.setAttribute("amenities", request.getParameter("amenities"));
  ```

  We count these operations as one FP per copied entity, and one FP per intermediate action in order to account for the overhead of implementing a whole new class. The only exception is again the last mask, for which no data preservation action is needed. In our example, we thus arrive at $(p-1) \times e + p - 1$ FP.
- **Entity retrieval:** Retrieving all entities for ultimate processing again requires $p \times e$ lookups (i.e. FP), this time from the session scope.
- **Session cleanup:** After retrieving all entities, they are not needed in the session scope anymore. We must therefore remove them as follows in order to prevent memory leaks:

  ```
  session.removeAttribute("type");
  session.removeAttribute("category");
  session.removeAttribute("amenities");
  ```

  Counting each removal as one function point, we have $(p-1) \times e$ statements (i.e. FP), since the entities provided by the last mask were not stored in the session scope.
- In total, we calculate 52 FPs for the *favorites* module in this approach.

The horizontal coupling between the individual masks is more loose here, since each is only concerned with its own data. However, cohesion remains low since we need a mask *and* and action for each dialog step. Vertical coupling between the application and data flow layer is tighter than in the first approach, since we need to take care of the data preservation and ultimate removal ourselves within the actions.

Table 1. Evaluation of data flow implementation strategies (FP for $p = 4, e = 3$)

	1. Data Flow Controller	2. Request Passing	3. Session Scope
Function Points	40	46	52
Horizontal Coupling	very loose (++)	very tight (−−)	loose (+)
Vertical Coupling	very loose (++)	very tight (−−)	tight (–)
Horizontal Cohesion	very high (++)	very low (−−)	low (–)
Vertical Cohesion	very high (++)	very low (−−)	low (–)

4.4 Summary

Table 1 summarizes the results of our discussion (for better readability, the qualitative criteria have been denoted with '+' for favorable and '–' for unfavorable constellations). As we can see, the approach employing the data flow control logic required the implementation of 13% less function points than the approach based on passing along request parameters, and 23% less FP than the approach using the session scope. The numbers of function points and thus the percentages will of course vary depending on the structure of the dialog graph and the contents of the dialog masks. However, the example scenario shows that significant savings in the volume of the implementation and specification are possible.

In contrast to the FP metric, the qualitative coupling and cohesion criteria apply to dialog graphs of any size and structure, since the technical steps that need to be taken for data propagation in each approach are independent of the graph structure. Our considerations of these criteria suggest that the use of the DFN and DCF enables very loose coupling and high cohesion, resulting in a more clearly structured implementation that should be easier to design, implement, test and maintain due to its lower complexity.

Note that we have assumed the use of the Dialog Flow Notation and Dialog Control Framework for *dialog flow* purposes in all three variants, in order to focus our discussion solely on the impact of our proposed approach to *data flow* control. A formal evaluation of the DCF's impact on the dialog flow aspects of web application development is beyond the scope of this article; however, we are certain that its use also considerably contributes to improving web applications' coupling and cohesion metrics.

5 Related Work

Many modeling languages for web-based applications have traditionally had a strong focus on data-intensive web applications [4], allowing developers to specify how users navigate through complex data schemas. More recently, a number of languages such as OOHDM [5], OO-H and UWE [6], and WebML [7] have also incorporated aspects of business process modeling, thereby narrowing the gap between process and navigation specifications that developers need to bridge. The data focus of these modeling languages is reflected in the variety of constructs they provide for specifying relations between and manipulations of data entities, most of which reside in the back-end of a web application.

However, apart from the transport links in WebML, which have similar semantics as our divergent data flows, the above modeling languages do not seem to provide explicit notation constructs for the fine-grained specification of inter-element data scopes and provision: The navigational model and process flow model in UWE, for example, specify how to navigate across and manipulate a data space provided by the back-end, but does not show which

data instances are provided from one interface component to another. In OO-H, activity and navigation access diagrams enable developers to specify which navigation nodes will invoke which data-manipulation methods on data objects, but any scoping of these instances is not explicitly modeled.

The extension to the Dialog Flow Notation we introduced here focuses on the fine-grained specification of data scopes and data provisioning (i.e. which data instances are made available to which dialog elements). These data flow specifications do not have to be implemented manually, but are automatically enforced by the Dialog Control Framework at run-time.

In contrast to the above notations, the DFN does not provide constructs for specifying how the data that is provided to the various dialog elements is manipulated. This is in keeping with our philosophy that the dialog flow is what distinguishes web applications most from traditional applications – we therefore focus the DFN on the typical and unique challenges of navigation and data flow in web applications, and encourage developers to use other established notations for modeling those aspects that go beyond this layer (e.g. by using activity or state diagrams to specify how data entities are manipulated in actions).

Regarding run-time support for complex dialog flows, popular web application frameworks have recently also adopted the notion of encapsulating dialog sequences in so-called "flows" (in Spring Web Flow [8]) or "conversations" (in the Shale Framework [9]), which have associated data scopes. However, these frameworks do not provide mechanisms for the realization of parallel or divergent data flows, and are lacking a corresponding notation that would provide executable specifications for such data flows.

6 Conclusion

In this paper, we presented a data flow extension to the Dialog Flow Notation (DFN) that enables web application developers to specify data flows between arbitrary elements in a dialog graph, as well as between nested dialog modules. Since all data flows are associated with events whose traversal is the condition for data propagation, and the receiver of a data flow may be different from the receiver of the associated dialog event, developers can build on the existing DFN semantics to specify fine-grained conditional data propagation within a web application's user interface. A new module-wide data scope complements the existing coarse scopes provided by common application servers.

To relieve developers from the tedious and error-prone effort of manually implementing secure and correct data flows throughout a web-based user interface, the data flow specifications created with this notation are executable: Using an Eclipse-based graphical editor, they can be transformed into machine-readable XML specifications that are interpreted by our Dialog Control Framework (DCF), which manages the dialog and data flow automatically. At run-time, the framework ensures that data entities are always provided to their specified receivers, and that dialog elements can only access those data entities that they are entitled to. Since the new data flow mechanisms are transparently added to the existing request scope, other frameworks relying on this scope can also benefit from the specified data flows.

Our work aims to support the development process for web-based applications in both the design and the development phase: We expect it to be easier for designers to communicate with clients and other non-technical stakeholders in the software development process, since data flow information that was previously contained only in business process models can now be

mapped to dialog flow diagrams, thereby providing a smoother transition from requirements to implementation. The comparison of the DFN with UML activity diagrams has shown that some data flow constructs that are particularly helpful in web application development cannot be modeled in UML, and the comparison with different data provision mechanisms indicated that the use of our data flow controller significantly contributes to reducing coupling and increasing cohesion in web applications. This allows stricter modularization and thus easier re-use of application components; not just on the level of the business logic, but all the way through the presentation logic. Given these considerations, we are confident that our model-driven approach to data flow specification and control can help developers to reduce web application implementation, testing and maintenance efforts.

Our implementation of the dialog and data flow control framework has already demonstrated that the described concepts are technically feasible. In our ongoing work, a first application for the data flow mechanism is the DiaGen extension to the DCF, which automatically breaks dialog masks into wizard-style interaction sequences at run-time to cater to different device capabilities [10] – the use of divergent data flows will greatly simplify the auto-generated data propagation code in this context. Apart from this, we are striving to test our hypotheses regarding the positive impact on the software development process in real-life projects. We are also considering further refinements of the data flow notation to provide interfaces to other modeling languages for the specification of the application logic's internal data handling. Another major ongoing research effort involving the data flow engine is the development of algorithms for handling backtracking in web applications.

Acknowledgements

We would like to thank Jan Richter for the implementation of the data flow controller.

References

1. M. Book, V. Gruhn, and J. Richter. Fine-grained specification and control of data flows in web-based user interfaces. In *Proc. 7th Intl. Conf. Web Engineering (ICWE 2007), LNCS 4607*, pages 167–181. Springer, 2007.
2. M. Gaedke, M. Beigl, H.W. Gellersen, and C. Segor. Web content delivery to heterogeneous mobile platforms. In *Advances in Database Technologies, LNCS 1552*. Springer, 1998.
3. M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proc. 19th IEEE Intl. Conf. Automated Software Engineering (ASE 2004)*, pages 100–109. IEEE Computer Society Press, 2004.
4. P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, Sep 1999.
5. G. Rossi, H. Schmid, and F. Lyardet. Engineering business processes in web applications: Modeling and navigation issues. In *3rd Intl. Work. Web-Oriented Software Technology*, pages 81–89, 2006.
6. N. Koch, A. Kraus, C. Cachero, and S. Meliá. Integration of business processes in web application models. *Journal of Web Engineering*, 3(1):22–49, 2004.
7. M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Software Engineering and Methodology*, 15(4):360–409, Oct 2006.
8. SpringSource. Spring web flow. http://www.springsource.org/webflow.
9. Apache Software Foundation. Shale framework. http://shale.apache.org.
10. M. Book, V. Gruhn, and M. Lehmann. Automatic dialog mask generation for device-independent web applications. In *Proc. 6th Intl. Conf. Web Engineering (ICWE 2006)*, pages 209–216. ACM Press, 2006.