Executable Semantics of Recursively Nestable Dialog Flow Specifications for Web Applications

Sören Blom, Matthias Book, Volker Gruhn
Applied Telematics/e-Business Group, Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany; Phone: +49-341-97-32330, Fax: +49-341-97-32339
{blom, book, gruhn}@ebus.informatik.uni-leipzig.de

Abstract

Information systems for the support of complex business processes are often equipped with web-based front-ends to allow convenient user access. To produce executable specifications of the users' interactions with such web-based applications, we use a visual language that enables developers to model their complex dialog structures. In this paper, we introduce the formal semantics of the core constructs of this Dialog Flow Notation: We define its syntax in terms of invariants about the permitted elements and their relations, and show how any words of the language (i.e. any syntactically correct dialog flow specifications) can be mapped to a deterministic pushdown automaton whose behavior defines the notation's semantics. This gives us and other tool developers a formal basis for the design and implementation of tools and frameworks that mirror the precise meaning of all DFN constructs.

1. Motivation

In the past years, web-based user interfaces have become increasingly popular front-ends for applications that shall be accessible anywhere, anytime, and on any device. Especially in the area of information systems that are designed to support complex business processes (both between enterprises, and between enterprises and their consumers), these applications have become increasingly sophisticated. The complexity of the business processes is typically mirrored by the complexity of the information system's navigation structure, which should enable users to perform their tasks efficiently, yet be flexible enough to deal with any contingencies that may occur within a process.

To model such complex navigation structures for web applications, we use the Dialog Flow Notation (DFN), a visual language for modeling the interplay between user activities and application operations that characterize a web application [4]. One of the core features of the DFN is the notion of "dialog modules" that can be nested arbitrarily in order to reuse dialog sequences in different contexts throughout an application.

While all DFN constructs were designed with careful regard to their conceptual and technical compatibility, the language's semantics have so far only been encoded operationally in the implementation of a Dialog Control Framework (DCF) that is capable of executing DFN-based dialog specifications. However, these semantics are not easily accessible to other tool developers, who thus cannot be completely sure that particular language constructs express exactly what they mean, or that framework implementations for other platforms work exactly as they intended. This has become especially apparent in our recent development of tools [3], applications [5] and extensions [6] based on the notation. The integration of tools for creating, validating and executing DFN specifications, and the parallel existence of several implementations of the framework, make precisely defined and well-understood semantics indispensable.

In this paper, we therefore introduce the formal semantics of the DFN's core constructs. After an informal overview of the main language features (Sect. 2), we formally introduce the elements of the DFN's syntax in terms of sets, relations and invariants that any DFN specification must conform to (Sect. 3). Then, we map this syntax onto an automaton model that realizes the run-time behavior the DFN describes (Sect. 4), and we show how these semantics can be implemented by tools in practice (Sect. 5). After an overview of the related work (Sect. 6), we discuss the benefits of these semantics for the implementation of tools and applications employing the DFN (Sect. 7).

2. Notation Overview

The DFN is a visual language for the specification of all possible user navigation steps and application reactions

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.25

(collectively termed the **dialog flow**) that can occur within a web-based application. As a running example for this section, we will refer to the dialog in Fig. 1 that models a typical scenario in web-based applications: If a user is not yet logged in, the application should ask him for his credentials, validate them and proceed according to the user's permissions – in a travel portal, for example, a regular user may be presented with forms to search for flights and hotels, while an administrator may see forms for updating the flight and hotel database.

2.1. Masks, actions and events

In the DFN, web pages are called **dialog masks** and symbolized by dog-eared sheets, while application logic operations are called **actions** and symbolized by circles. Collectively, we call these basic elements **atomic elements** since their implementation is not relevant at the level of the navigation, which is the sole focus of the DFN. In Fig. 1, for example, the *login* mask may contain a form for users to enter their credentials, and the *check credentials* action may contain logic to validate that input.

In the implementation, developers may be tempted to blur the distinction between masks and actions, since it is technically possible to implement both presentation and application logic in either component. To realize the clean separation of presentation and application logic that the DFN encourages, developers are urged to model all components that generate page markup as masks, and all components that do not generate markup as actions.

These elements are connected by **dialog events** (symbolized by arrows) that specify which masks or actions are called under which conditions. For any event, this condition depends on which element generated it, and what label it carries: In our example, the *ok* event is received by the *mark user as logged in* action if it was generated by the *check credentials* action, but received by the *has admin rights?* action if generated by the *mark user as logged in* action. For events generated by masks, the label is determined by a particular parameter in the HTTP request (typically identifying the link or button that the user clicked, such as the *submit* event created by the *login* mask in our example). For events generated by actions, the label is set by the application logic (typically identifying the outcome of the operation, such as the *incorrect* event generated by the *check credentials* action).

Every dialog element can generate and receive multiple events (only one at a time, though), enabling the developer to draw complex **dialog graphs** that specify all possible transitions between masks and actions. Since we can easily conceive useful dialog graph fragments comprising several consecutive masks or actions, the DFN does not require dialog graphs to be bipartite (in Fig. 1, for example, splitting

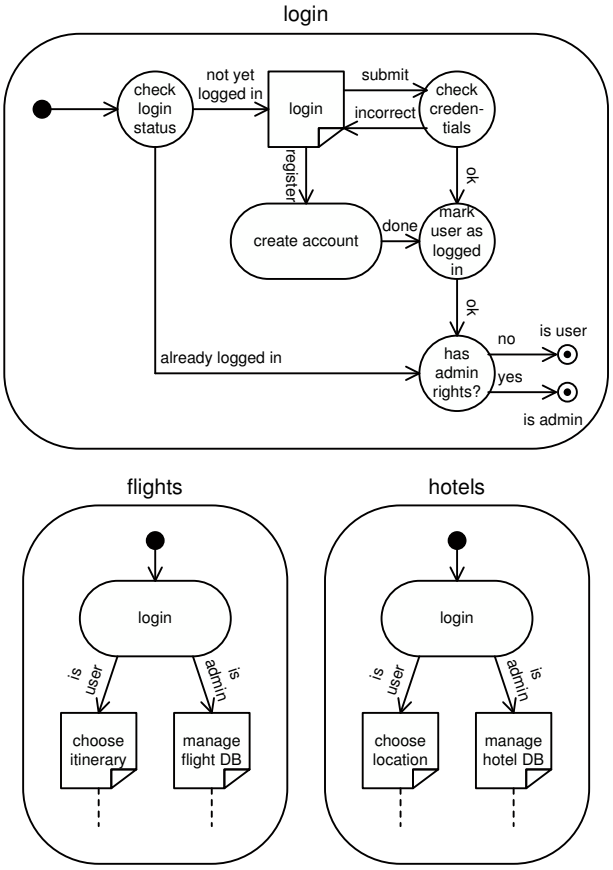

Figure 1. Definition of (top) and references to (bottom) the *login* dialog module in a travel portal.

the processing of the user's credentials into three separate actions enables the module to react flexibly to different situations, like bypassing the credential check when the user is already logged in). This implies that not all DFN events are equivalent to HTTP requests or responses – rather, a mask-to-mask transition implies a full request-response cycle, while an action-to-action transition happens completely on the server within a larger request-response cycle.

2.2. Dialog modules

Theoretically, the complete dialog flow of an application could be described using only atomic elements. However, the resulting specifications would quickly become too large to handle conveniently, and the "flat" structure does not support reuse of often-needed dialog graphs. Since reuse of program parts is a fundamental concept in programming and makes particular sense in user interaction, where the user expects to perform similar tasks in similar ways, the

DFN provides a construct to encapsulate dialog graphs in **dialog modules** that can then be invoked from other dialog graphs. To maximize the potential for re-use, dialog modules can call each other recursively – thus, they are not mere meta-constructs for structuring the designer's view of the dialog graph, but dialog elements with run-time semantics in their own right.

Conceptually, we need to discern between a module's **definition**, which specifies the "interior" dialog graph contained in a module, and a module's **references**, which show how a module is embedded in other (its so-called "exterior") dialog graphs. Module definitions are symbolized by large boxes with round corners containing the interior dialog graph, while module references are symbolized by smaller oblate ovals. When talking about the nesting relationships between modules, we use the term "sub-module" to denote a module whose reference is nested into another module's definition, and the term "super-module" to denote a module whose definition contains references to other modules. Figure 1, for example, shows the definition of the *login* module, which contains a reference to the *create account* sub-module, and is itself referenced in the *flights* and *hotels* super-modules. The interior dialog graphs of the *flights* and *hotels* modules thus are exterior dialog graphs of the *login* module.

When a module is called from an exterior dialog graph that it is embedded in, traversal of its interior dialog graph begins at the **initial anchor** (symbolized by a solid disk). When the traversal of the interior dialog graph reaches a **terminal anchor** (symbolized by a circled dot), the module terminates, and an event originates from it that continues the traversal of the exterior dialog graph. This so-called "terminal event" carries the same label as the terminal anchor that had just been reached. For example, the *is user* and *is admin* terminal anchors in the *login* module definition correspond to the *is user* and *is admin* events that originate from its reference in the *flights* and *hotels* super-modules.

In the DFN, all dialog graphs must be encapsulated in modules that call each other. At the top of this invocation hierarchy must be a **root compound** whose traversal starts when a client sends its first request to the application server (e.g. requesting the home page). The root compound is a special type of dialog module since it only has a definition (symbolized by a thicker contour), but no references to it. Also, since there is no way for users to "terminate" a web application (as they can only leave the site, e.g. by closing the browser window), the root compound cannot meaningfully terminate. Hence, we prohibit that it contains any terminal anchors, and instead recommend that its interior dialog graph should be cyclic.

2.3. Need for formalization

The above overview of the DFN may suffice as an informal introduction to give developers a general idea of the involved concepts. However, to enable application developers to model dialog graphs that precisely express their expectations of the web application's behavior, and to enable tool developers to implement dialog validation and control algorithms that work precisely as prescribed by the specifications, we need to define our language's semantics formally.

While the features introduced above represent only a subset of the DFN, these transitions between masks and actions, as well as the nesting of dialog modules, are the essential concepts that form the foundation for all other language constructs. As we will see in the following sections, these concepts already constitute a basic language in their own right, whose formalization is a non-trivial challenge.

The formalization effort comprises two steps: Firstly, we need to define the DFN's syntax, i.e. formally express the types of its elements and the invariants that must hold for all dialog flow specifications, i.e. all words of the DFN. Secondly, we need to define the DFN's semantics, i.e. formally express the behavior of any application described by a word of the DFN. These steps will be presented in detail in the following two sections.

3. Formal Syntax of the DFN

While the Dialog Flow Notation is a visual language, we do not describe its visual syntax (i.e. its icons and their relationships such as connectedness, insideness etc.) here. Rekers and Schürr, for example, nicely show how to separate the different layers of visual syntax, and how to arrive at a conceptual representation (the abstract syntax graph) [21]. We do not show this straightforward step here, as we are not concerned with the visual peculiarities of our language, which we regard as quite conventional in this regard. Hence, we will introduce the sets and relationships directly from the notation on the level of the abstract syntax graph.

In the following subsections, we introduce a number of functions to express relationships between elements of the language. Unless otherwise noted in the function's definition, we assume that all these functions are total. Strictly speaking, we would have to require this property by formulating invariants on the functions. However, since dialog graphs that do not satisfy these invariants would already be incomplete and thus nonsensical on the visual level (e.g. because they contain "dangling" events or unlabeled elements), we rely on the prerequisite that these constraints are already ensured by the visual syntax.

3.1. Dialog elements

The syntax of the DFN can be expressed in terms of invariants that must hold for sets containing the elements we introduced in the previous section:

Atoms and anchors. For the formal model, we first define the finite sets of all dialog masks (E_{mask}), actions (E_{act}), initial anchors (E_{init}) and terminal anchors (E_{term}) of an application.

Note that the labels that dialog elements carry in the visual notation are just a feature allowing humans to conveniently refer to individual elements. Conceptually, the elements' identities are derived from their existence as separate visual entities with distinct spatial coordinates, which are independent of the labels visually associated with those entities (for example, a mask x and an action x in a module m, and a mask x in a module n all carry the same label, but constitute three distinct entities). Consequently, we do not define a mapping of elements to labels here, as we do not associate any semantics with this mapping anyway.

Modules. Dialog modules differ from atoms and anchors in that they appear in two forms, namely their definition and use. In order to distinguish the two forms, we define:

Definition 1 *E_{mod} is the finite set of all module references within an application.*

Definition 2 *D_{mod} is the finite set of all module definitions within an application.*

In the visual notation, the labels written inside module reference icons and above module definition contours associate references with their respective definitions. In the formal model, we can represent this association more immediately by a reference relation that indicates which module reference refers to which module definition:

Definition 3 *The total function $\tau : E_{mod} \to D_{mod}$ is the reference relation that specifies which module definition $d \in D_{mod}$ a module reference $e \in E_{mod}$ refers to.*

Root compound. As we have seen, the root compound is a special element insofar as it shall not be nested into (i.e. called from) any other module. We therefore define it explicitly as an element that does not belong to the codomain of the reference relation τ, so no module reference can refer to it:

Definition 4 *$d_0 \notin D_{mod}$ is the root compound.*

Since the root compound shares some properties that we will define for nestable modules, we introduce a super-set of all modules that comprises both types:

Definition 5 *$D := D_{mod} \cup \{d_0\}$ is the finite set of all module definitions and the root compound definition.*

To simplify the formulation of some of the following constraints, we subsume the different dialog element types in the sets of all atomic elements (E_{atom}), all elements able to generate events (E_{gen}), all elements able to receive events (E_{rec}), and all elements in the whole application (E):

Definition 6 *The finite sets*

$$\begin{aligned} E_{atom} &:= E_{mask} \cup E_{act} \\ E_{gen} &:= E_{atom} \cup E_{mod} \cup E_{init} \\ E_{rec} &:= E_{atom} \cup E_{mod} \cup E_{term} \\ E &:= E_{atom} \cup E_{mod} \cup E_{init} \cup E_{term} \end{aligned}$$

are unions of their pairwise disjoint subsets.

Containment constraints. With this groundwork laid, we can focus on the relationships between the element types we just defined. First of all, we define a function that associates all dialog elements with the module definition that they are contained in:

Definition 7 *The total function $\mu : E \to D$ is the containment relation that specifies which module definition $d \in D$ contains an element $e \in E$.*

Multiplicity constraints. While developers may place an arbitrary number of masks, actions and module references in a module definition's interior dialog graph, the placement of anchors must obey special rules. As we have seen, every module must have exactly one entry point. As modules also need to terminate at some point to return control to the super-module from which they were called, every module must also have at least one exit point. The root module, however, cannot have any terminal anchors, since it has no super-module to return to.

Invariant 1 *Every module, as well as the root compound, must have exactly one initial anchor:*

$$\forall d \in D : \exists_1 e \in E_{init} : \mu(e) = d$$

Invariant 2 *Every module must have at least one terminal anchor:*

$$\forall d \in D_{mod} : \exists e \in E_{term} : \mu(e) = d$$

Invariant 3 *The root compound must not have any terminal anchors:*

$$\forall e \in E_{term} : \mu(e) \neq d_0$$

3.2. Dialog events

Since dialog events do not represent physical entities, but just transient input, we do not need to represent each of their visual instances as a unique set element. Instead, the set of events just contains all unique event labels found in a model (including the "empty label" ε):

Definition 8 *$V \supset \{\varepsilon\}$ is the finite set of all events within an application.*

Receiver relation. The connection of elements through events is expressed by the so-called receiver relation. This is a partial function since not all elements can generate all events – rather, every element will typically generate only a small subset of all events:

Definition 9 *The partial surjective function*

$$\eta :\subseteq E_{gen} \times V \rightarrow E_{rec}$$

is the receiver relation that defines which element $e' \in E_{rec}$ receives the event $v \in V$ generated by the element $e \in E_{gen}$.

By defining the function as surjective, we have already ensured that every receiver is reachable. In addition, valid dialog flow specifications are subject to a number of additional constraints:

Regular event constraints. To enforce the encapsulation of dialog graphs in modules, we first require that events connect only elements within the same module – it is not allowed to specify an event from an element within one module to an element within another module:

Invariant 4 *The generator and receiver of any event must be in the same module:*

$$\forall((e, v), e') \in \eta : \mu(e) = \mu(e')$$

To ensure that the traversal of dialog graphs is deterministic, we require that any element cannot generate two events with the same name:

Invariant 5 *Any two events generated by the same element must be different from each other:*

$$\forall((e, v), e'_1), ((e, w), e'_2) \in \eta : v \neq w$$

To ensure that a module can be successfully traversed, the DFN requires that there must be a path from the initial anchor to each receiver in a module, and a path from each generator to a terminal anchor. These constraints can easily be formulated using the transitive closure of the receiver relation - however, since η is not a homogeneous relation, we first need to project it into a relation with equal domain and codomain:

Definition 10 *The partial function*

$$n :\subseteq E \rightarrow E,$$
$$n = \{(e, e') \in E_{gen} \times E_{rec} \mid \exists v \in V : \eta(e, v) = e'\}$$

is the connection relation indicating that $e \in E_{gen}$ and $e' \in E_{rec}$ are directly connected by at least one event.

We can now use the transitive closure n^+ of this relation to require that all elements in the graph are reachable, and that there are no dead ends:

Invariant 6 *Any receiver must be reachable on a path from the initial anchor:*

$$\forall e' \in E_{rec} : \exists e \in E_{init} : (e, e') \in n^+$$

Invariant 7 *At least one terminal anchor must be reachable on a path from any generator, except in the root module:*

$$\forall e \in E_{gen} : \quad \mu(e) \neq d_0 \Rightarrow \exists e' \in E_{term} : (e, e') \in n^+$$

Initial event constraint. To ensure the unambiguity of the dialog graph's starting point, exactly one event (the so-called "initial event") must be generated by each module's initial anchor. Since its traversal does not depend on the receipt of an event generated by a preceding mask or action, it must be unlabeled (in contrast to any other event, which must carry a label in order to be identified at run-time):

Invariant 8 *Initial events, and only these, must be unlabeled:*

$$\forall((e, v), e') \in \eta : e \in E_{init} \Leftrightarrow v = \varepsilon$$

Terminal event constraints. Terminal anchors must carry information on how the traversal of the exterior dialog graph shall continue upon termination of a module. They are therefore associated with the event that shall continue traversal of the module's exterior dialog graph upon its termination:

Definition 11 *The total function $\lambda_{term} : E_{term} \rightarrow V \setminus \{\varepsilon\}$ is the terminal event relation that specifies which event $v \in V \setminus \{\varepsilon\}$ a terminal anchor $e \in E_{term}$ is associated with.*

To guarantee consistency of a module's interior and exterior dialog graphs, the labels of the terminal anchors in its interior dialog graph must match the labels of its terminal events in any exterior dialog graphs – the DFN does not allow more, less or other events to originate from the module reference than there are terminal anchors in the module's definition, and vice versa.

Invariant 9 *For all modules, the terminal events they generate must all be associated with the terminal anchors they contain:*

$$\begin{aligned}\forall e \in E_{mod}: \quad & (\exists v \in V, e' \in E_{rec} : \eta(e, v) = e' \\ \Leftrightarrow \quad & \exists t \in E_{term} : \mu(t) = \tau(e) \wedge \lambda_{term}(t) = v)\end{aligned}$$

Words of the DFN language. Given these definitions and invariants, we can now define any word in the Dialog Flow Notation as a tuple:

Definition 12 *An application's dialog flow is defined by*

$$\begin{aligned}\mathcal{D} := (E_{mask}, E_{act}, E_{init}, E_{term}, E_{mod}, \\ D_{mod}, d_0, \tau, \mu, V, \lambda_{term}, \eta)\end{aligned}$$

4. Notation Semantics

Based on the above syntactic foundation, we can now describe the semantics of the DFN. In Sect. 2, we stated that the purpose of the DFN is to specify the behavior of the dialog control logic so that it accepts a certain subset of user-generated input events depending on the current state in the dialog flow, and generates a response by triggering the invocation of masks and/or actions. To achieve this, we need to map the DFN specification to a suitable automaton that models the run-time behavior we aim to describe through the DFN. Given that, we can explain the semantics of the DFN as specifications of such an automaton.

4.1. Suitable model for run-time behavior

At first sight, one might believe that a finite state machine is sufficient for our task, since the atomic elements can simply be interpreted as states and the events as transitions in a state transition graph. Modules might be considered as simple constructs for refining certain states, so that by resolving all nesting levels, a very large but flat transition graph could be derived. Yet, if one considers the capability to recursively embed modules into each other, it quickly becomes obvious that we need at least a (deterministic) pushdown automaton (DPDA).

We follow the reasoning of Green [12], who speaks of *transition networks* as special adaptations of DPDAs – the crucial difference is that the transition network is augmented with an output function referred to as the "action function". In automata theory, the output function γ is usually omitted, as decidability is of interest. For dialog models, however, an automaton without output, i.e. without responses to user input, would be useless, so we include it in our automaton model.

In general, a deterministic pushdown automaton $\mathcal{P}$ is a tuple $\mathcal{P} = (Q, \Sigma, \Omega, \Gamma, \delta, \gamma, q_0, Z_0, F)$, with:

- Q, the finite set of states
- Σ, the finite set of input symbols
- Ω, the finite set of output symbols
- Γ, the finite set of stack symbols
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$, the transition function
- $\gamma : Q \to \Omega$, the output function
- $q_0 \in Q$, the initial state
- $Z_0 \in \Gamma^*$, the initial stack string
- $F \subseteq Q$, the set of accepting states

A configuration of a DPDA $\mathcal{P}$ is described by $(q, S) \in Q \times \Gamma^*$, where q is the current state of $\mathcal{P}$ and the string $S = s_1 s_2 \cdots s_n$ (an element of the Kleene closure Γ^*) denotes the contents of the stack. A transition to a subsequent configuration (q', S') occurs when $\mathcal{P}$ receives an input $i \in \Sigma$, so that $\delta(q, i, s_1) = (q', S')$, where S' is the string that should replace s_1 on top of the stack (for example, $S' = s_0 s_1$ would indicate that s_0 is pushed on top of s_1, while $S' = \varepsilon$ would indicate that s_1 is removed from the top of the stack). After completion of a transition, the new state is q', and the stack content is $S' s_2 \cdots s_n$. Having identified the new state q', the output $o \in \Omega$ is produced by evaluating $\gamma(q') = o$.

Such a DPDA represents a dialog system in the following way: Σ represents the spectrum of possible input symbols coming in from the user or application, and Ω represents the available reactions (e.g. dialog masks to be displayed or application logic to be executed). Whenever the user or the application provides input that can be recognized by the DPDA, a transition occurs. At its end, a new output operation is invoked that will provide new input, etc.

4.2. Mapping DFN words to DPDAs

We will now discuss the individual elements of the DPDA tuple $\mathcal{P}$ and show how they are constructed from the individual elements of a DFN tuple $\mathcal{D}$.

States. Following a bottom-up approach, we will first discuss which language elements of the DFN should be considered as states in a DPDA. Obviously, atoms resemble states, so we will include them in Q. Regarding the anchors, one might argue that initial and terminal anchors are just notation constructs, rather than actual states of the application. However, we will see that these states are transiently assumed in the process of calling and terminating dialog modules, and thus need to be in Q. That leaves us with module references. While they are embedded into the dialog graph just like atoms, they do not represent actual states but serve as references to module definitions. Thus, the first "tangible" state related to a module is its initial anchor, which we have already noted for inclusion.

In addition, we have to include an element that has no direct analogy in the DFN; an error state. The automaton falls back to this state, denoted by †, when an input symbol is read for which $\mathcal{D}$ does not define an event in the current context. Altogether, we get:

Definition 13 $Q := E_{atom} \cup E_{init} \cup E_{term} \cup \{\dagger\}$ *is the DPDA's set of states.*

Input alphabet. At run-time, the dialog control logic can receive input from masks or actions in the form of events. This is the only kind of "input" we can specify within the DFN, so the set of input symbols corresponds to the set of events.

Definition 14 $\Sigma := V$ *is the DPDA's set of input symbols.*

Output alphabet. We interpret "output" as the information on which mask or action is invoked next. Hence, the atoms constitute the output alphabet:

Definition 15 $\Omega := E_{atom}$ *is the DPDA's set of output symbols.*

Stack alphabet. DFN words specify the nesting and termination of modules at run-time. In order to keep track of the order in which modules are nested at any given time, we need to put them on a stack that reflects the nesting hierarchy. Hence, we equate the stack alphabet of the DPDA with the set of module definitions.

Definition 16 $\Gamma := D$ *is the DPDA's set of stack symbols.*

Initial state and stack symbol. Defining the initial state means to define the "entry point" into the dialog flow. In the DFN, we defined the root compound as the entity that cannot be nested into any other module. We also defined the initial anchor of any module to be the starting point of its dialog graph. Hence, we define the initial anchor of the root compound to be the initial state of the DPDA, and consequently the root compound to be the initial stack symbol:

Definition 17 $Z_0 := d_0$ *is the DPDA's initial stack symbol.*

Definition 18 $q_0 \in \{e \mid e \in E_{init} \wedge \mu(e) = d_0\}$ *is the DPDA's initial state.*

The latter definition is unambiguous and always possible since Inv. 1 requires that there is exactly one initial anchor per module, and Def. 4 establishes that there is exactly one root compound for each DFN word.

Accepting states. When an automaton is used for language recognition, arrival at an accepting state (after reading the complete input sequence) indicates whether this sequence is part of the language or not. This notion does not exactly fit the scenario of a web-based dialog controller, where no finite input sequence exists *a priori*. Here, an input sequence can only be deemed "invalid" if an input symbol is encountered that was not expected in the current state, i.e. if the respective event was not specified in the DFN's receiver relation η. In this situation, we said, the system would assume the error state †. This state, together with the initial and terminal states (that are of a transient nature only) are the non-accepting states, leaving the atoms as accepting states:

Definition 19 $F := E_{atom}$ *is the DPDA's set of accepting states.*

Transition function. Finally, we need to define the transition function. The challenge here is to exactly map the transitions between dialog elements in the DFN, and the nesting and termination of modules, to matching transitions in the DPDA. In the following paragraphs, we will define the transition function $\delta(e, v, d)$ for a dialog element $e \in Q$, an event $v \in \Sigma$ and a module definition $d \in \Gamma$ as the disjoint union of several independent functions that together cover the different types of transitions in the dialog flow.

We will illustrate each step using the simple example in Fig. 2, which shows a recursive dialog flow on top and the corresponding DPDA on the bottom. In the DPDA, we annotate transitions in the form $v, d/d^*$, where v is the incoming event, d the top stack element, and d^* is the string of elements to replace the top stack element. The initial anchor states are denoted by $INIT_m$, where m is the name of the respective module definition.

Transitions to atoms. The most straightforward transition is an event that leads to an atom, i.e. $\eta(e, v) \in E_{atom}$. Since we remain within the same module, the stack of the DPDA remains unchanged:

Definition 20 *The transition to an atom is defined by*

$$\delta_{\succ atom}(e, v, d) = (\eta(e, v), d)$$

In the example, we can see this e.g. in the transition $\varepsilon, W/W$ between the initial anchor state $INIT_W$ of the root compound and the action state A.

Transitions to sub-modules. If an event leads to a submodule (i.e. $\eta(e, v) \in E_{mod}$), the module reference delivered by η cannot be used as the new state, as discussed above. Rather, we need to push the called module's definition onto the stack and use its initial anchor as the new state:

Definition 21 *The transition to a module reference (i.e. the calling of a module) is defined by*

$$\delta_{\succ mod}(e, v, d) \quad = \quad (i \in E_{init} : \mu(i) = \tau(\eta(e, v)), \\ \tau(\eta(e, v))d)$$

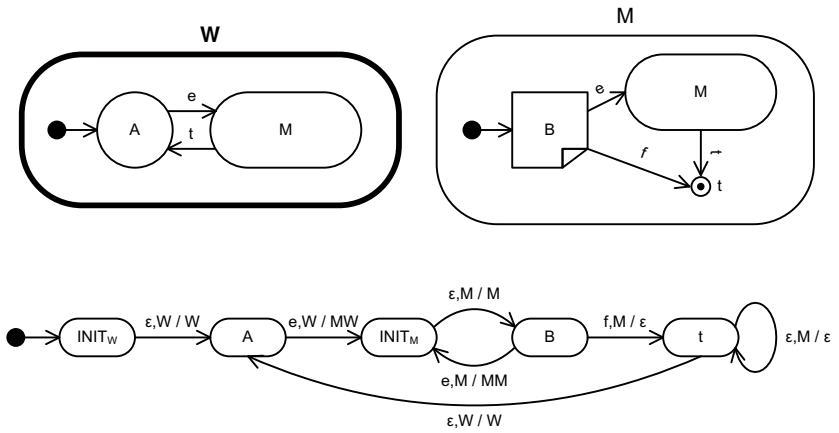

Figure 2. A recursive DFN example (top) and its translation into a DPDA (bottom).

The example illustrates this e.g. in the transition $e, W/MW$ between the action state A and the initial anchor state $INIT_M$ of the module M.

Transitions to terminal anchors. If an event leads to a terminal anchor (i.e. $\eta(e, v) \in E_{term}$), the module containing the anchor will be terminated, and the dialog flow continues with the receiver of the terminated module's terminal event in the super-module's dialog graph. This occurs in two steps – in order to identify the super-module, we must first remove the current top element from the stack:

Definition 22 *The transition to a terminal anchor (i.e. the termination of a module) is defined by*

$$\delta_{\succ term}(e, v, d) = (\eta(e, v), \varepsilon)$$

In the example, this step occurs in the transition $f, M/\varepsilon$ from the mask state B to the terminal anchor state t.

Transitions from terminal anchors. After the transition to the terminal anchor has removed the top stack element, the DPDA can now determine the transition from the terminal anchor (i.e. $e \in E_{term}$) to the receiver of the terminal event. For this purpose, we find the sub-module m within the now-current module definition d (i.e. $\mu(m) = d$) that the terminal anchor e was contained in (i.e. $\mu(e) = \tau(m)$), as well as the event v' that is associated with the terminal anchor (i.e. $\lambda_{term}(e) = v'$). We can then retrieve this event's receiver (i.e. $\eta(m, v')$), while the current module d remains unchanged:

Definition 23 *The transition from a terminal anchor (i.e. from a terminated module's reference) is defined by*

$$\begin{array}{lcll} \delta_{term\succ}(e, v, d) & = & (\eta(m, v'), d) & \\ & & \textit{with } m \in E_{mod}: & \mu(m) = d \wedge \\ & & & \mu(e) = \tau(m) \\ & & \textit{and } v' \in V: & \lambda_{term}(e) = v' \end{array}$$

The example shows this e.g. in the transition $\varepsilon, W/W$ from the terminal anchor state t to the action state A.

Complete transition function. To complete the transition function's construction, we combine the above partial functions and add a transition to the error state † for all cases where a new state could not be determined otherwise:

Definition 24 *The PDA's transition function is*

$$\delta(e, v, d) = \begin{cases} \delta_{\succ atom}(e, v, d) & \textit{for } \eta(e, v) \in E_{atom} \\ \delta_{\succ mod}(e, v, d) & \textit{for } \eta(e, v) \in E_{mod} \\ \delta_{\succ term}(e, v, d) & \textit{for } \eta(e, v) \in E_{term} \\ \delta_{term\succ}(e, v, d) & \textit{for } e \in E_{term} \\ (\dagger, d) & \textit{otherwise} \end{cases}$$

We can easily show that the combination of these functions together cover the whole set of transitions that may occur for any dialog flow specification:

- δ covers all inter-element transitions explicitly specified in η, since the receiver sets E_{atom}, E_{mod} and E_{term} are exactly those sets whose union constitutes the codomain of the DFN's receiver relation.
- δ covers all inter-module transitions implied in η, since the case $e \in E_{term}$ is the only situation in which we need to find a successor for a state that is not in the domain of η.
- δ covers all remaining transitions that are undefined in η, since all the valid alternatives are handled by the above branches.

Output function. In Sect. 4.1, we argued that we need an output function to let the DPDA meaningfully model the desired application behavior, where the output symbols are

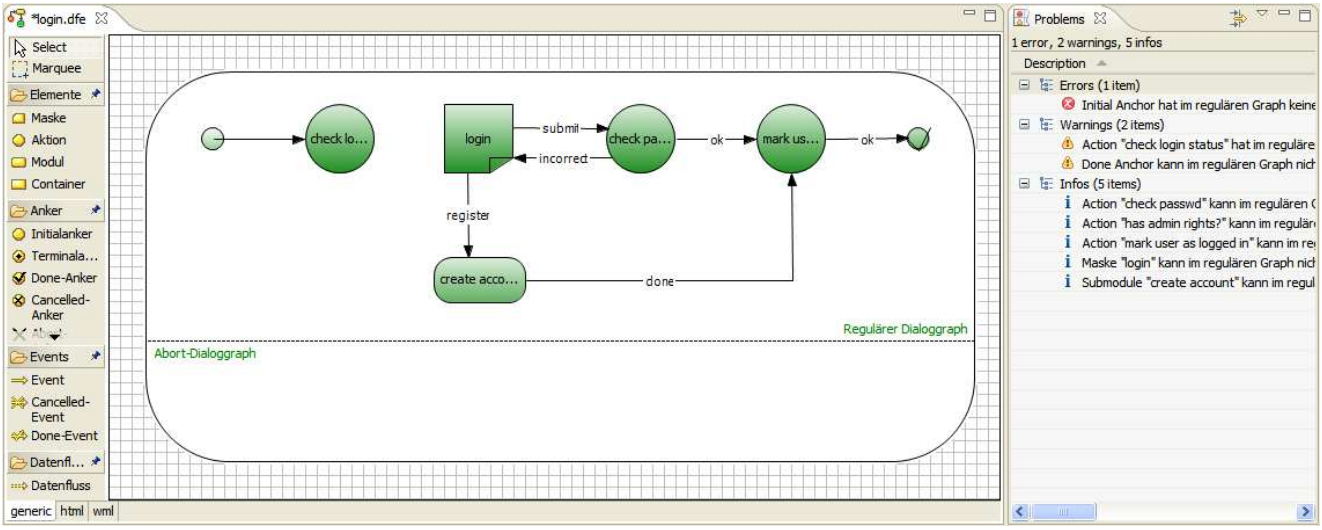

Figure 3. Eclipse-based Dialog Flow Editor with validation results.

the masks and actions. In contrast to a classic Mealy machine, the output depends only on the current state and occurs immediately upon entering the state.

When defining the states of the DPDA and the transition function, we have seen that it was necessary to introduce some "transient" states that handled the nesting and termination of modules. Those states do not generate any kind of output (other than "empty output" ε). In the definition of the output function, we therefore need to distinguish between states that are atoms and those that are not:

Definition 25 *The DPDA's output function γ is defined by*

$$\gamma(q) = \begin{cases} q & \text{for } q \in E_{atom} \\ \varepsilon & \text{otherwise} \end{cases}$$

With this, we have shown how a DPDA $\mathcal{P}$ can be constructed from any DFN word $\mathcal{D}$ as a formal model of the interpretation of the specification $\mathcal{D}$ by a suitable dialog control logic. Besides providing precise rules for the implementation of the run-time logic, the formal representation is also helpful in checking the correctness of dialog flow models, as we will show in the following section.

5. Support for Tool Development

Having introduced the formal syntax and semantics of the DFN, we will now show how these formalisms support the development of corresponding tools for web application development. As an example, our current tool chain consists of a visual editor for the specification of dialog flows, a validation component for checking dialog flow models for syntactic and semantic correctness, and a runtime framework driving the execution of web applications' dialog masks and actions according to a specified dialog flow model.

The dialog flow editor shown in Fig. 3 enables application developers to build dialog flow models from visual shapes corresponding to the elements of the Dialog Flow Notation. As this editor has been realized as a plug-in for the Eclipse IDE, it uses the Eclipse Modeling Framework (EMF) for the internal representation of all model elements, where instances of `Mask`, `Action`, `Module` classes etc. correspond to the elements of the sets E_{mask}, E_{act}, E_{mod} etc.

To give application developers instant feedback on the correctness of their model, the editor employs a validation component that continually checks the model's consistency and indicates any constructs violating the DFN rules.

Regardless of how the visual representation is implemented in a particular editor, it can easily be validated and ultimately executed if it corresponds to the sets and relations defined in Sect. 3. Our dialog flow validator, for example, transforms the visual editor's EMF model into a Prolog fact base by 1:1 translation of dialog elements into `mask`, `action`, `module_ref` etc. facts, and dialog events into `event` facts. Simple relationships between elements can be expressed by rules such as the following, which reflects part of Def. 6:

```
receiver(ID) :-
  mask(ID); action(ID);
  module_ref(ID); terminal_anchor(ID).
```

On this basis, we can check the model's validity by simply formulating the invariants introduced in Sect. 3 as Prolog rules, and running them against the Prolog prover to find any violating elements.

As an example, the following Prolog rule checks Inv. 6 (every receiver must be reachable from an initial anchor):

```
violates_inv06(ID) :-
  receiver(ID),
  \+((init_anchor(IA), path(IA, ID))).
```

For this rule, the Prolog prover will return all IDs of receivers for which there exists no initial anchor on a path to them. The IDs of these elements are then returned to the editor, which displays error or warning messages depending on the severity of the violation, as shown in the right-hand panel of Fig. 3. In total, we implemented about 20 rule checks in this way to cover all constructs of the DFN (including a number we could not discuss in this paper for the sake of brevity).

This approach enabled a very straightforward implementation of the validator component, since we did not need to be concerned with the technicalities of e.g. traversing references in an object-oriented graph structure, but could work on the actual level of the notation semantics, as the formal invariants and the Prolog rules are very similar. In consequence, we can be sure that the validator component enforces the notation's precise semantics. This is a major benefit of the formal semantics, since e.g. an object-oriented implementation of informal prose semantics would be more prone to errors in the design of its rule-check algorithms, as well as their technical realization.

The formal DPDA model lends itself to implementation in a dialog control logic that mirrors the semantics described in Sect. 4. At the core of our Java Servlets-based Dialog Control Framework (DCF) is a stack-based automaton model similar to the one shown in Fig. 4. For each user, it maintains the current configuration $(q, S) \in Q \times \Gamma^*$ in the `DialogAutomaton`'s `currentElement` attribute (representing q) and its stack of `DialogModules` (representing S). While we distinguished module references (E_{mod}) and definitions (D_{mod}) in the formal model, the object-oriented design enables us to model their relation more efficiently by representing both in the same `DialogModule` instance: Each of these instances holds mappings (corresponding to the receiver relation η) of generating and receiving `DialogElements` associated by their connecting `DialogEvents`, and each `DialogModule` instance can be referenced as such a generator or receiver itself. The `DialogEvent` and `DialogElement` instances constituting these dialog graphs are created upon initialization of the framework based on the previously validated dialog flow specifications, which can be expressed e.g. in XML format.

At run-time, the `handleEvent` method then refers to these mappings in order to implement the behavior of the DPDA's transition relation δ, using the `push`, `pop` and `top` methods to work with the module stack, and the `invokeAtom` method to call masks and actions, as defined by the DPDA's output function γ.

As the example of our tool chain shows, the Dialog Flow Notation has executable semantics: Any dialog flow model composed of the sets and relations defined above can be automatically validated for its syntax according to the given invariants, and any dialog flow model that satisfies these invariants can be automatically executed by a run-time environment based on a DPDA. The close alignment of the language's theoretical foundation and its practical specification, validation and execution is beneficial for tool developers as it provides a more solid and more rigorously testable basis for their implementation. This also benefits application developers, who can be more certain that a web application will actually behave according the intention they expressed in their model.

6. Related Work

Research on UI design has been dealing with models for dialog-based systems since over 20 years. For example, Olsen described in 1984 how to use a PDA for user interaction management [19]. Shortly after, Green published his widely acknowledged survey which compares different dialog models [12]. The approach we present here follows his idea on formalizing recursive transition networks as DPDAs, and applies it to the field of web-based applications.

Many approaches exist to specify dialogs for "classical" (i.e. window-based) GUIs on desktop systems: Kleyn and Chakravarty use event decomposition graphs [17] (an adapted version of AND/OR graphs) to specify semantics of dialogs; Jacob works with nested state diagrams [15]. Both are concerned with rather low-level aspects of dialogs, such as mouse movements, yet abstract from layout concerns. Goedicke and Sucrow suggest graph grammars as a specification method for specifying dialogs in (classical) GUIs [11]. In their approach, dialogs (together with their internal structure) are modeled as graphs, and transitions between graphs describe the possible changes of UI objects. Berstel et al. introduce the concept of "visual event grammars" [2]. They model dialogs by sets of automata that represent parts of a GUI and communicate through events, but focus mainly on the interplay of events and objects (i.e. widgets) within individual dialogs.

All of these approaches require rather complex formalizations, as they either include very finely-grained event semantics (mouse movement, scrolling) or aim at describing the sequence of dialogs together with the structure and behavior of widgets inside dialogs. In our opinion, this is not just a consequence of the higher complexity of desktop systems' GUIs, but rather a strong indication that the dialog flow should be engineered and designed separately from the the presentation and application logic's internals.

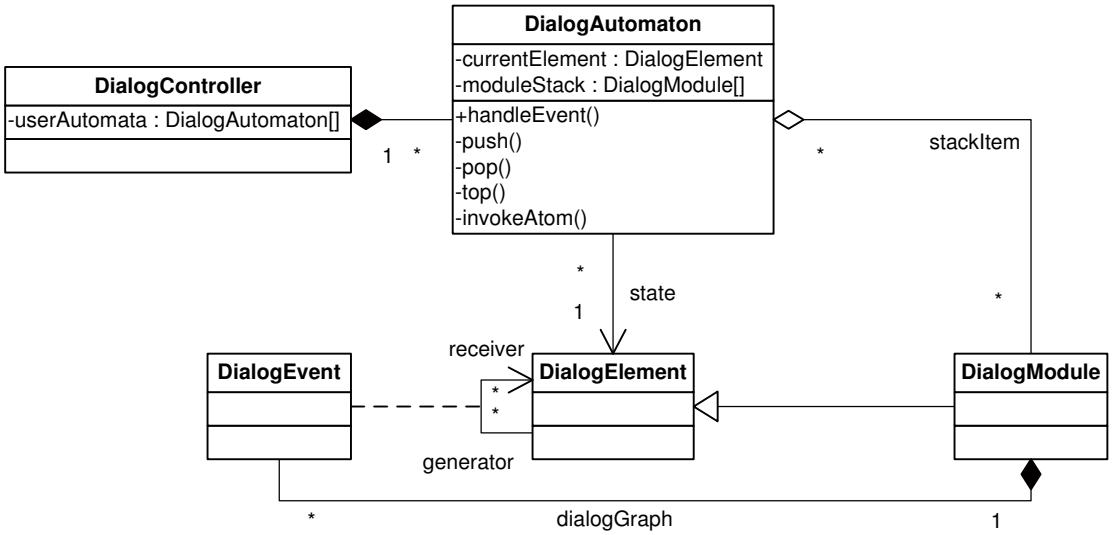

Figure 4. Dialog controller design.

The notion of a dialog graph in our sense was used by Britts in his survey of different UI systems and approaches available at that time [8]. His dialog graphs share many semantical concepts with the dialog flow of the DFN, but lack the additional separation between masks and actions that the DFN employs to model the core components of web-based applications.

Ariav and Calloway pursue a similar approach with their Dialog Charts [1]. They focus on the "outside" aspects of dialogs, i.e. their sequence, and also separate user interaction from machine interaction. Yet, their approach does not comprise constructs for nesting dialog modules, as the DFN does, and is not accompanied by a formalization of the notation.

In the realm of hypertext-based applications, Statecharts are often used, such as in the approach of Oliveira et al. [10], and the Hypercharts extension proposed by Paulo et al. [20]. Whilst concerned with the flow of web pages, these specifications focus to a great extent on the synchronization of stateful document parts, such as multimedia content, which is not the focus of the DFN.

The FARNav approach [13] resembles the DFN in its explicit focus on how application operations (in addition to user activities) influence a web application's dialog flow, but lacks means to represent embeddable modules, as most of the aforementioned Statecharts-based models. The intention and utilization of Statecharts in Winckler and Palanque's StateWebCharts approach [22] comes closest to the concepts of the DFN, and is provided with a formal definition of the semantics.

WebML [9] is another approach for comprehensive modeling of the data structure, presentation and navigation structure in web applications, with particular strength in applications with dynamic data access. The fine-grained data flow control mechanisms provided by the DFN [7] can be formalized by mapping them to interprocedural data flow graphs [14]; an approach which is beyond the scope of this paper, however.

One might also consider coloured Petri nets [16] or UML activity diagrams (as used in UWE [18]) suitable for the formulation of dialog graphs. While these approaches can be used to model the basic dialog graphs we have shown in this paper, some more complex dialog patterns such as inheriting dialog graph fragments from generic presentation channels, or aborting, resuming and returning from dialog modules with run-time identification of unspecified event receivers [4] would turn out quite cumbersome to model using these languages.

For easy adoption in practice, we therefore designed the Dialog Flow Notation to be similar to established languages in its broad concepts, but tailored to the challenges of web-based dialog flows in its detailed constructs. Our rationale for the choice of a deterministic pushdown automaton in the formalization is similar: While one could conceive a number of suitable formalisms for defining the precise syntax and semantics of the DFN, we chose the DPDA since it is most illustrative of how an actual implementation of the DFN's semantics in a framework could work.

7. Conclusions

In this paper, we described the formal semantics of the Dialog Flow Notation's core features, which can serve as an authoritative basis for tool developers who need to reason formally about such dialog flow specifications in the process of validating or executing them. Using examples

from a dialog editor-validator-controller tool chain, we have shown the benefits that such executable semantics provide in terms of the clarity of corresponding implementations.

The semantics of the DFN features presented here can also serve as a benchmark for the expressive power that any new constructs will contribute: Depending on the formalization level that will be required to define them (i.e. within the visual syntax, the conceptual syntax or the behavioral semantics), we can identify which features are mere syntactic sugar and which add more expressive power, and thus get an indication of the complexity of the necessary validation and execution logic.

Since a web application cannot be completely specified in terms of its dialog flow only, we have previously introduced additions to the notation for specifying the presentation rules for dialog masks on different devices [6], as well as for specifying data flows between presentation and application logic [7]. We are currently integrating the formal semantics of those modeling perspectives with the navigation model described in this paper.

As we have shown, the DFN's formal semantics are useful for the implementation of tools on a wide range of platforms ranging from Java Servlet containers to Prolog theorem provers. In our continuing work, we will investigate the applicability of these semantics to other execution environments for dialog-intensive applications. Besides considering established platforms for web-based applications such as PHP, Ruby etc., we are especially interested in applying these semantics to other interaction paradigms (as enabled by AJAX) or architectural patterns (e.g. Rich Internet Applications, distributed or mobile services etc.), and examining the feasibility of their efficient implementation based on the same formal model.

8. Acknowledgments

The implementation of the dialog flow editor, validator and framework was part of a joint project with itCampus GmbH in Leipzig, supported by a technology support grant from the European Regional Development Fund (ERDF) 2000-2006 and funds of the Free State of Saxony. The Applied Telematics/e-Business group is endowed by Deutsche Telekom AG.

References

[1] G. Ariav and L.-J. Calloway. Designing conceptual models of dialog: A case for dialog charts. *SIGCHI Bull.*, 20(2):23–27, 1988.

[2] J. Berstel, S. C. Reghizzi, G. Roussel, and P. S. Pietro. A scalable formal method for design and automatic checking of user interfaces. In *23rd Intl. Conf. on Software Engineering (ICSE 2001)*, pages 453–462. IEEE Computer Society, 2001.

[3] S. Blom, M. Book, V. Gruhn, and R. Laue. Switch or struggle: Risk assessment for late integration of COTS components. In *2nd Intl. Workshop on Incorporating COTS Software into Software Systems, co-located with the 29th Intl. Conference on Software Engineering*. IEEE Computer Society Press, 2007.

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

[5] M. Book and V. Gruhn. Experiences with a dialog-driven process model for web application development. In *Workshop on Model Driven Agile Development II – Component-Based Agile Development, co-located with the 29th Annual Intl Computer Software and Applications Conf. (COMPSAC 2005)*, pages 173–178. IEEE Computer Society, 2005.

[6] M. Book, V. Gruhn, and M. Lehmann. Automatic dialog mask generation for device-independent web applications. In *6th Intl. Conf. on Web Engineering (ICWE 2006)*, pages 209–216. ACM Press, 2006.

[7] M. Book, V. Gruhn, and J. Richter. Fine-grained specification and control of data flows in web-based user interfaces. In *Web Engineering - 7th Intl. Conference (ICWE 2007), Lecture Notes in Computer Science (LNCS 4607)*, pages 167–181. Springer Verlag, 2007.

[8] S. Britts. Dialog management in interactive systems: a comparative survey. *SIGCHI Bull.*, 18(3):30–42, 1987.

[9] S. Comai and P. Fraternali. A semantic model for specifying data-intensive web applications using WebML. In *Semantic Web Workshop, Stanford*, 2001.

[10] 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.

[11] M. Goedicke and B. E. Sucrow. Towards a formal specification method for graphical user interfaces using modularized graph grammars. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA, 1996. IEEE Computer Society.

[12] M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, Jul 1986.

[13] M. Han and C. Hofmeister. Modeling and verification of adaptive navigation in web applications. In *6th International Conference on Web Engineering (ICWE 2006)*, pages 329–336. ACM Press, 2006.

[14] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994.

[15] R. J. K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Trans. Graph.*, 5(4):283–317, 1986.

[16] K. Jensen. Coloured petri nets: A high level language for system design and analysis. *Advances in Petri Nets 1990, LNCS*, 483:342–416, 1991.

[17] M. F. Kleyn and I. Chakravarty. Edge - a graph based tool for specifying interaction. In *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*, pages 1–14, New York, NY, USA, 1988. ACM Press.

[18] N. Koch and A. Kraus. The expressive power of UML-based web engineering. In *Second International Workshop on Web-oriented Software Technology (IWWOST02)*, pages 105–119. CYTED, 2002.

[19] D. Olsen. Pushdown automata for user interface management. *ACM Trans. Graph.*, 3(3):177–203, 1984.

[20] F. B. Paulo, P. C. Masiero, and M. C. F. de Oliveira. Hypercharts: Extended statecharts to support hypermedia specification. *IEEE Trans. Softw. Eng.*, 25(1):33–49, 1999.

[21] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[22] M. Winckler and P. Palanque. StateWebCharts: A formal description technique dedicated to navigation modelling of web applications. In *Interactive Systems: Design, Specification, and Verification: 10th International Workshop, DSV-IS 2003: Revised Papers*, number 2844 in LNCS. Springer, 2003.