FORMS2DIALOG: AUTOMATIC DIALOG GENERATION FOR WEB TASKS

Nobal B. Niraula[1],*Amanda Stent*[2],*Hyuckchul Jung*[3],*Giuseppe Di Fabbrizio*[4],*I. Dan Melamed*[5],*Vasile Rus*[1]

[1] Department of Computer Science, The University of Memphis, TN, USA
[2] Yahoo! Labs, New York, NY, USA
[3] AT&T Labs, Bedminster, NJ, USA
[4] Amazon.com, Cambridge, MA, USA
[5] AT&T Labs, New York, NY, USA

{nbnraula,vrus}@memphis.edu[1], stent@yahoo-inc.com[2]
hjung@research.att.com[3], pino@difabbrizio.com[4], {lastname}@research.att.com[5]

ABSTRACT

Today, many common tasks (*e.g.* booking flights, ordering food) can be done by filling out web forms. Automatic processing of Web forms to support interactive speech input is useful for numerous reasons, including ease of use for mobile device users and accessibility for people with visual or print disabilities. In this paper, we propose an automated method to process web forms and convert them into dialog flows for spoken interaction. First we identify relevant information for each form element (including element type, label, values and help messages) and key relationships between form elements (including ordering and dependencies). We then generate two types of dialog flow for each Web form. Experimental results show that the method generates efficient and informative dialog flows for web tasks, a key step for building virtual assistants. An Android application has been realized as a use case of the generated dialog flows.

Index Terms— Web, Dialog Systems, Virtual Assistants

1. INTRODUCTION

The Web has become an integral part of our daily lives. We can now perform many common tasks (*e.g.*, booking flights, buying products, checking train schedules, searching news) by interacting with web forms. Web forms range from simple one-element search forms to multi-element, multi-page forms. Figure 1 shows a sample form. Currently, virtual assistants such as *Siri*[1] , *Google Now*[2] and *Cortana*[3] are becoming popular alternatives to web browsers. Given a dialog flow for a task, virtual assistants are able to use spoken interaction with users to accomplish the task. A key research problem for virtual assistants is therefore automatic generation of dialog flows corresponding to common web tasks.

[1]http://www.apple.com/iphone/features/siri.html
[2]http://www.google.com/landing/now
[3]http://www.windowsphone.com/en-us/how-to/wp8/cortana/meet-cortana

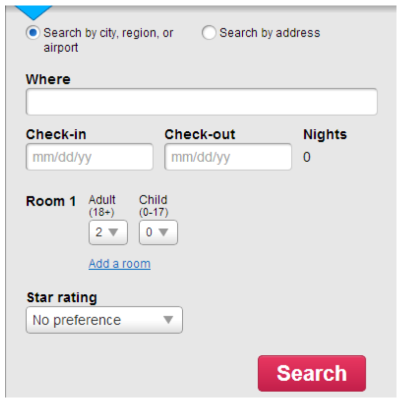


Fig. 1. Web form example: Orbitz.com

In this paper, we present a method for automatically generating dialog flows from web forms to support spoken interaction. Our method automatically processes a web page to identify relevant information for each form element (including element type, label, values and help messages), and identifies key relationships between form elements (including ordering, value constraints and dependencies). We model a dialog flow extracted from a web form as a slot-filling dialog; a partially ordered sequence of form elements (slots), each identified by a label, a type (*e.g.* check box, selection list), and optionally a set of input values and/or help messages. This permits automatically generated dialog flows to be refined using methods such as reinforcement learning or supervised learning over interaction logs (*e.g.* [1, 2]). It also permits the use of standard markup languages for voice interaction. In our work, the dialog flows are automatically rewritten at run-time into FXML, a dialog markup language used by the Florence spoken dialog manager [3][4].

The automatic generation of dialog flows for web tasks

[4]This work was done while the first five authors were at AT&T Labs.

is applicable in many situations. Currently, typing into web forms is the main method by which tasks are executed on mobile devices. While typing may be easy for most people wih a regular-sized keyboard, it becomes a real challenge with a small-screen mobile device. As a result of automatic dialog flow generation, users can instead interact with web forms using spoken or multimodal dialog. Automatically generated dialog flows can also improve accessibility for people with visual or print disabilities.

2. RELATED WORK

Automatic processing of web forms is challenging for several reasons. First, the structure of web forms is developer dependent. For example, there are many travel websites, each with a different distribution of task information across web pages and forms, different form layouts, different auxiliary tasks (*e.g.* hotel booking, car rental reservation), and even different form element labels (*e.g. Depart* vs. *From*). Second, web forms are optimized via CSS for visual interaction – the HTML source may not reflect the layout, appearance, and order of form elements [4]. Third, web form elements may be dynamically generated or modified by javascript or AJAX scripts based on user input; complete information about these forms does not appear in the static source code of the web page. Fourth, the source code of web pages is often incomplete or incorrect; for example, not all web page developers use the optional HTML `<label>` tag to specify the actual name of a form element.

There are three types of approach to acquiring task flows from the web. The first type is *show and tell* [5]; a software agent is trained by a user who shows and simultaneously describes the steps needed to complete the task, and then the software agent is asked to repeat the steps to accomplish other instances of the same or similar tasks. These systems need sample demonstrations from an expert, and the number of demonstrations required increases with the task complexity.

The second type of approach does not require a user to explicitly describe what they are doing, but does require multiple examples, *e.g.* as click-stream data. This approach is mostly commonly used to generate *web accessibility models* for people with visual or print disabilities (*e.g.* [6]). This approach has the advantage of being entirely automatic, but does require many click streams to learn an interaction model; furthermore, it does not learn the user utterances and system prompts necessary for dialog interaction.

The third type of approach generates task models directly from web forms with no access to examples of human interaction with the web form (*e.g.* [7, 8]). Our approach belongs to this category. Given a URL pointing to (a) web form(s), we create task representations in a fully automated way. We then use the Florence dialog manager to allow the user to interactively perform the task using speech input [3].

The work of Barbosa *et al.* [9] is most closely related

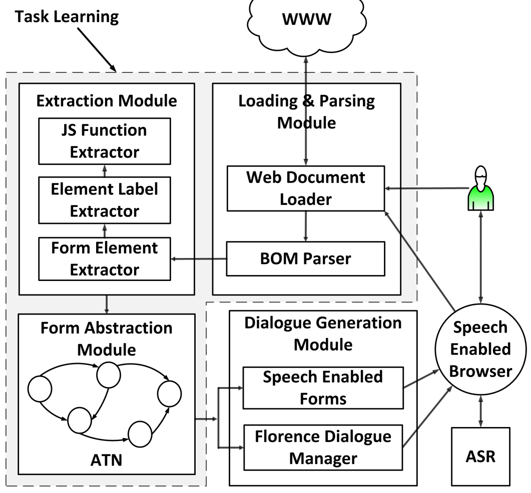


Fig. 2. F2D architecture

to ours. They proposed a system to semi-automatically generate or select language models for free-text form elements. Our work complements theirs: we automatically identify the possible paths through sets of form elements (dialog flows), the values of non-free-text form elements, and form element labels and help messages.

In older work, Issar [10] proposes a speech interface for forms on the web. This system loads the URL given by its user, identifies the form elements in it, and helps the user to fill out the forms. The user speaks the labels/values of the form or provides the values by typing/clicking. Issues with this approach include: the user has to mention slot labels explicitly and can only fill out one slot at a time; there is no error handling; and there is no way to optimize the interaction. The system provides speech input, but not dialog.

3. SYSTEM OVERVIEW

The architecture of our system, called F2D (Forms2Dialog), is shown in Figure 2. It consists of four modules. First, the user provides the URL(s) of one or more web pages containing web form(s) that implement the target task. The *loading and parsing module* is responsible for downloading one or more web pages from each starting URL supplied by the user and parsing the web page(s) using an HTML parser. The parsing process generates the DOM and BOM of each web page. The DOM is simply a tree of HTML elements. The BOM provides the true state of the HTML elements as they would appear in a browser window.

The *extraction module* extracts forms and form elements from input web page(s). It extracts both static (always present) and dynamically generated form elements (those that appear in a form as a result of user input; *e.g.* if a user chooses "multi stop" in a flight booking form, an AJAX script may add additional fields to the form for each stop). It automatically extracts for the type and label of each form element, and potential values and help messages (if available in the form). Finally, it automatically finds ordering and dependency relationships between form elements. We detail the methods used in our extraction module in Section 4.

The *form abstraction module* expresses form content as an Augmented Transition Network (ATN) [11]. An ATN consists of states and transitions. States in our ATNs represent form elements, and the transitions represent possible user actions. Different ATNs can be designed for the same web form (see Section 5). ATN transitions can be further weighted or modified through user interaction logs or reinforcement learning, through we do not address this in this paper. Finally, the *dialog generation module* generates dialog flows for web forms using ATNs. Since our dialog manager is Florence, it rewrites ATNs into FXML. Florence is a dialog manager developed at AT&T [3]. It uses a declarative XML-based language, Florence XML (FXML), to represent spoken dialog flows. An FXML document consists of states (which can contain variables and other metadata), transitions, conditions (on states), and global variables or registers. The current input and local context are used to control the interaction flow. Each state can also specify operations to be executed while entering or exiting the state; these can be used to invoke other networks or subdialogs, return control to a previous dialog, or affect the task or global state (*e.g.* to perform database lookup). Since task acquisition involves identifying subtasks and slots for each subtask, Florence is a natural choice for our work. We give more detail about the form abstraction and dialog generation modules in Section 5.

4. FORM EXTRACTION AND LABELING

In this section, we present our approach to form extraction for web tasks. Since our approach is completely automated, we need a tool that simulates human interactions with Web forms. We use jQuery, a popular JavaScript library, and Selenium[5], a Java-based software testing framework for web applications that allows programmatic simulation of user interactions.

4.1. Form Element Extraction

As mentioned previously, the input to F2D is the URL(s) of web page(s) from which the dialog flow is to be acquired. We start by extracting all the forms in the page corresponding to an input URL. Then we extract form elements from each form. A form typically contains elements of several types, *e.g.* text box, text area, radio button, check box, select (list selection), other buttons (*e.g.* submit, reset) and hyperlinks (F2D follows hyperlinks to extract multi-page forms). Form elements can be visible or invisible (invisible elements are used to hold user information and preferences, and information computed automatically). A form can also contain events that can activate or deactivate other form elements. We store the visual and layout information associated with each form element including its visibility, its absolute location, its location relative to other form elements, its font, size and colour, and the browser order of form elements (the "tab order"). We also extract default form element values, possible form element values (*e.g.* for radio button, check box, and submit elements), and help messages, where available.

[5]http://seleniumhq.org/projects/webdriver/

4.2. Form Element Labeling

Form element labeling is a key dialog flow generation task; labels provide the core of system prompts. However, as we discussed in Section 2, accurate extraction of form element semantics and form element contents is challenging. Motivated by prior work in this area [7, 8, 6], we use structure, visual and layout features to associate input form elements and labels. We propose three methods (M_1, M_2 and M_3) for the labeling task using these features.

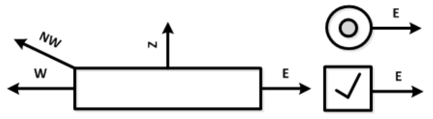


Fig. 3. Heuristics for BOM-based form labeling

Method M1: Method M1 uses only the `for` attribute of the HTML <label> tag, which allows web developers to associate text labels with input form elements. This method provides high precision. However, since the `for` attribute of the <label> tag is not mandatory and many web developers do not use it, this method gives low recall for most web forms.

Method M2: As mentioned earlier, web forms are optimized for visual interaction through particular browsers. The HTML source captures the static content of the web page, but may not reflect its visual layout or appearance or dynamic page content. However, no matter whether server side or client side scripts are used to generate web page content, web form elements are eventually displayed to the user when the page is loaded in the browser. The browser's internal representation, or BOM, thus contains the most accurate and up-to-date information about the visual appearance and dynamic content of the web page. Method M2 uses the visual information from the BOM to predict input form element labels. In particular, we treat each visible text element around an input form element as a potential candidate for its label. Moreover, we observe that labels corresponding to text box, text area and select form elements are (in English web pages) normally located either to the north (above), west (left) or north-west of the input form element, whereas labels corresponding to check boxes and radio buttons are normally located to their east (right) (see Figure 3). We consequently assign to each input form element the visible text elements closest to it, using the relative location (left, right, etc.) as a tie breaker.

Method M3: Method M3 uses the DOM, a hierarchical tree-based representation of the elements in the page. This method uses field- and segment-scopes for the labeling as in Furche *et al.* [7][6]. In field-scope labeling, we start from each web form element and traverse towards the root of the web form. We stop when we find an ancestor that contains other form element(s) than the target element. We then record all the text nodes in the sub-tree of the ancestor we just visited and assign these as labels of the element. In segment-scope labeling, we cluster the form elements using their style information to form a segment tree. For this, we follow the algorithm mentioned in [7] but use the class and style attributes of the elements to check style equivalent nodes. The clusters are then utilized to label elements that didn't get a label in field-scope labeling.

4.3. Finding Constraints and Dependencies

We observed two major ways in which input form elements influence each other: value constraints and dependencies. *Value constraints* are when the (partial) value provided for one form element changes the possible values of other input form elements. For example, starting to type in a text field may cause a drop down list of possible values for that text field to appear, and to change with additional typed input; selecting a car manufacturer from a list of makes in a car reservation form may change the values for the model field. *Dependencies* are when additional form elements appear, or become visible, based on user input; for example, selecting "round trip" in a flight booking form may cause a return date/time field to appear. Figure 4 highlights these in the Orbitz.com web form from Figure 1.

If form element values are listed in a web form, then we store them with the form element. F2D, through calls to Selenium, can also simulate interactions with web form elements (*e.g.* can "type" into text fields or "select" check boxes and radio buttons) and then scan for changes in the BOM for the web page. If changes are found, we store the value constraints or dependencies in the ATN representation of the task.

4.4. Evaluation of Form Extraction and Labeling

We evaluated our form extraction and labeling pipeline on the ICQ data set[7], a popular, if somewhat old, dataset for this kind of evaluation. The data set consists of 20 web pages in five different domains (airfare, auto, book, job and real estate). The form elements in all web pages in this data set are manually annotated for type and label.

In Table 1 we report results for form element extraction and labeling for this data set. We measured performance in

[6]Note that we do not apply Furche *et al.*'s method for layout scope labeling or their domain-specific templates.

[7]http://metaquerier.cs.uiuc.edu/repository/datasets/icq

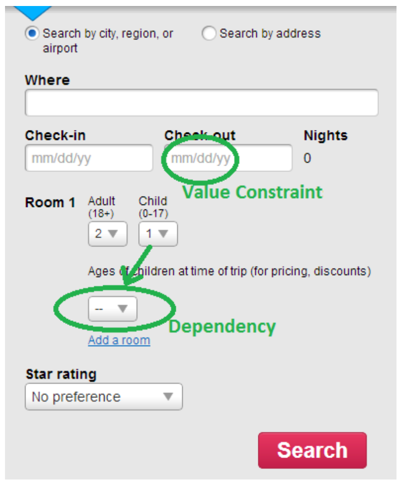


Fig. 4. Value constraint and dependency in the Orbitz.com web form

Table 1. *Form labeling accuracies (in %) for M1, M2 and M3 and their combinations in ICQ data set*

	M_1	M_2	M_3	M_1-M_2-M_3	M_1-M_2-M_3
Airfare	42.18	70.14	68.72	72.51	75.35
Auto	3.09	83.50	85.56	85.56	88.65
Book	0.9	84.15	78.21	89.10	85.14
Job	5.6	53.93	58.42	67.41	67.41
Real Est.	15.57	59.01	53.27	61.47	59.83
All	13.49	70.15	68.84	75.21	75.28

terms of accuracy (the number of correctly extracted and labeled fields divided by the total number of manually labeled fields) for M1, M2, M3, M1-M2-M3 and M1-M3-M2. To get a label for an element in M1-M2-M3, we use the label found by M1 (if available), the label found by M2 if M1 fails, and the label found by M3 if both M1 and M2 fail. The combination M1-M3-M2 is defined similarly.

The performance of M1, as expected, is the lowest among all the methods. The reason is that M1 relies on the labels voluntarily specified by web developers - these are very accurate but also rare. Accuracy is the reason we preferred M1 first when combining methods. M2 and M3 achieve comparable performance to each other; however, the combinations of all three methods achieve highest performance, indicating that M2 and M3 provide somewhat complementary information. Our best combination of methods achieves overall macro-averaged accuracy of 75.2%. Accuracy is high for the Airfare, Auto, and Book domains. For the Job and Real Estate domains, there were missing CSS files for some of the web forms in the data set which caused a drop in performance.

By comparison, for field labeling on the ICQ dataset Dragut *et al.* [4] report an overall accuracy of 92% for their method, and of 80% for the alternative WISE method, and

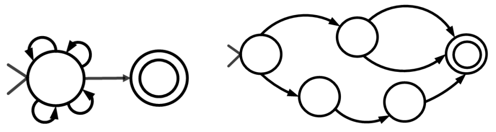

(a) Two state (b) Exhaustive expansion

Fig. 5. Dialog process representations

Furche *et al.* [7] report an F measure of 96%. Our methods are more simple and general than theirs, and by using the BOM we are able to deal with more modern approaches to web development than are exhibited in the ICQ data set, such as AJAX-heavy dynamic forms; however, we can incorporate some of the rules of Dragut *et al.* in our system in the future.

We could not assess accuracy of constraint and dependency extraction, as these are not labeled in the ICQ data.

4.5. Demo of Form Extraction

We have implemented a simple interface to F2D that allows the user to provide a URL for a web form to be run through F2D, and to inspect the output ATN, which can be drawn using graphViz[8]. The user may then choose to interact with the corresponding extracted dialog flow using the F2DGo mobile application, as described in the next section.

5. DIALOG FLOW EXTRACTION

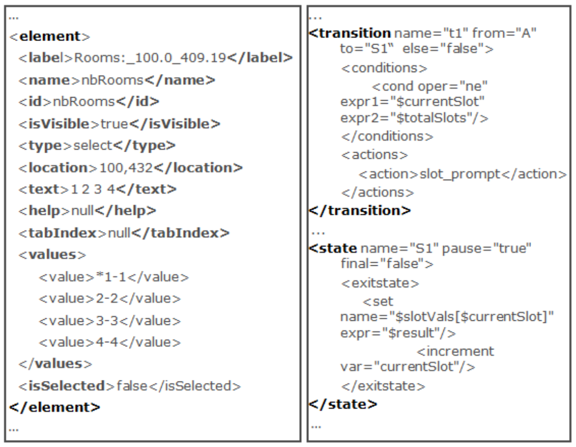
F2D supports two types of dialog flow: two-state, and exhaustive expansion. These are depicted in Figure 5.

Two State Model (TSM): This very simple dialog flow representation consists of only two states: the start state and the accept state. The form elements' metadata (types, labels, values, and help messages) is stored in the global context, outside of the ATN. The dialog system repeatedly selects a form element, prompts for it using its label, and collects a value, until all the input form elements have values; then, the transition to the accept state is executed. This representation is very general and flexible, but cannot represent value constraints or dependencies, or a default ordering over input form elements. We use it because it is most similar to the information state model of dialog [12]. Optimal orderings over form elements could be learned from user interaction logs or through reinforcement learning (*e.g.* [1, 2]); we leave this for future work.

Exhaustive Expansion Model (EEM): This representation has a state for each form element, and transitions following the browser ordering (the "tab order"). Each state stores the metadata (type, label, values, and help messages) for its form element. Value constraints are represented as constraints on states. Dependencies are represented as sub-ATNs. This representation stores more of the dialog flow structure; however, there may still be multiple possible dialog flows, and as with the two-state model, we leave the acquisition of preferences over these possible dialog flows for future work.

[8]http://www.graphviz.org

ATNs extracted by F2D are stored as XML files as shown in Figure 6. This information is used to generate FXML dialog flows corresponding to the two-state and exhaustive expansion models. A fragment of a generated FXML dialog corresponding to a two-state model is also presented in Figure 6. In F2D we use the Florence dialog manager to support spoken interaction with these dialog flows.



```
...
<element>
  <label>Rooms:_100.0_409.19</label>
  <name>nbRooms</name>
  <id>nbRooms</id>
  <isVisible>true</isVisible>
  <type>select</type>
  <location>100,432</location>
  <text>1 2 3 4</text>
  <help>null</help>
  <tabIndex>null</tabIndex>
  <values>
     <value>*1-1</value>
     <value>2-2</value>
     <value>3-3</value>
     <value>4-4</value>
  </values>
  <isSelected>false</isSelected>
</element>
...
```

```
...
<transition name="t1" from="A"
     to="S1" else="false">
     <conditions>
          <cond oper="ne"
     expr1="$currentSlot"
     expr2="$totalSlots"/>
     </conditions>
     <actions>
        <action>slot_prompt</action>
     </actions>
</transition>
...
<state name="S1" pause="true"
     final="false">
     <exitstate>
        <set
     name="$slotVals[$currentSlot]"
     expr="$result"/>
              <increment
     var="currentSlot"/>
     </exitstate>
</state>
...
```

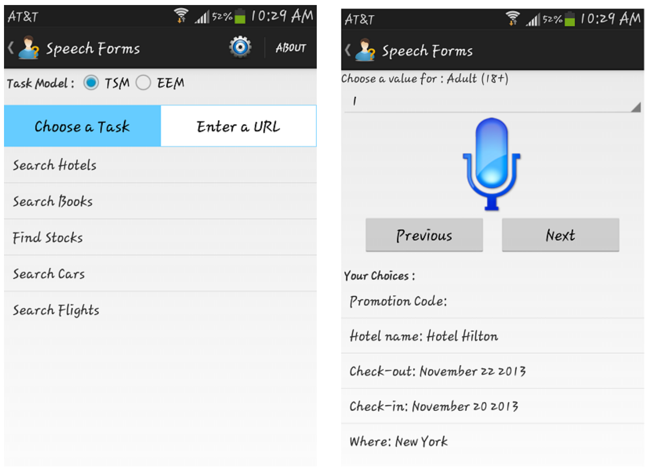
Fig. 6. Form information (*left*) and FXML fragment (*right*)

5.1. Demo of Dialog Flow Extraction

We cannot evaluate dialog flow generation against a standard data set, because there is no standard data set of dialog flows for web tasks. However, we have implemented a prototype mobile dialog system to informally assess extracted dialog flows. The system, called F2DGo, is an Android web application supporting speech and multimodal interaction with dialog flows through the Florence dialog manager. Interaction with F2DGo takes place in three stages: task choice, slot filling, and results presentation.

In the **task choice** stage, the user can select a dialog flow (a two-state or exhaustive expansion model) and can then choose from a set of listed tasks pre-extracted from web forms by F2D (*e.g.* Search hotels) or provide a URL from which F2D will obtain a dialog flow for a task. These choices are sent to F2D, which extracts a web form if necessary, and returns a dialog flow as an FXML file.

In the **slot filling** stage, F2DGo executes the selected / generated dialog flow using speech and text prompts. The extracted form information is used to determine which form elements to prompt for, the order in which to prompt for them, and the prompts themselves. For example, if the form element is a text field, the system constructs a query from the form element label. On the other hand, if the element is a radio button or list of options, the system prompts using the list of



Task Model : [✓ TSM] EEM

Choose a value for : Adult (18+)

1

Your Choices :

Promotion Code:

Hotel name: Hotel Hilton

Check-out: November 22 2013

Check-in: November 20 2013

Where: New York

(a) The menus (b) Multi-modal Interaction

Fig. 7. Snapshots of the prototype mobile application

choices the user can select from. The user may provide form element values using speech, typing or GUI selection. When the user has supplied values for all required form elements in a dialog flow, F2DGo sends them to F2D.

In the **results presentation** stage, F2D automatically fills out the web form from the original URL using the user-supplied values, and returns a link to the results page to F2DGo. F2DGo currently just displays this results page. However, it could be processed to extract individual result records as done by [13, 14, 15]; we leave this for future work.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we described a method for acquisition of dialog flows from web forms to support interactive speech input. Key features of our method are: (a) it is entirely automatic; (b) it is fast and scalable; and (c) it is accurate and very general. In addition to presenting and evaluating our web form extraction method, we present a prototype mobile application that uses the extracted dialog flows to support speech-centric mobile interaction.

As future work, we plan to implement methods to extract result sets from HTML lists and tables. We also plan to experiment with statistical methods for optimizing extracted dialog flows to deal with speech recognition and understanding errors, *e.g.* supervised learning from interaction logs, or reinforcement learning from a user simulation. Finally, we plan to experiment with more complex interactions involving interleaved information gathering and results presentation stages.

7. REFERENCES

[1] S. Lee, “Structured discriminative model for dialog state tracking,” in *Proceedings of SIGDIAL*, 2012.

[2] O. Pietquin, M. Geist, S. Chandramohan, and H. Frezza-Buet, “Sample-efficient batch reinforcement learning for dialogue management optimization,” *ACM Transactions on Speech and Language Processing*, vol. 7, no. 3, 2011.

[3] G. Di Fabbrizio and C. Lewis, “Florence: a dialogue manager framework for spoken dialogue systems,” in *Proceedings of INTERSPEECH*, 2004.

[4] E. Dragut, T. Kabisch, C. Yu, and U. Leser, “A hierarchical approach to model web query interfaces for web source integration,” in *Proceedings of VLDB*, 2009.

[5] J. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, and W. Taysom, “PLOW: A collaborative task learning agent,” in *Proceedings of AAAI*, 2007.

[6] J. Mahmud, Y. Borodin, I.V. Ramakrishnan, and C.R. Ramakrishnan, “Automated construction of web accessibility models from transaction click-streams,” in *Proceedings of WWW*, 2009.

[7] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, and C. Schallhart, “Forms form patterns : Reusable form understanding,” in *Proceedings of WWW*, 2012.

[8] R. Khare and Y. An, “Understanding deep web search interfaces: a survey,” in *Proceedings of SIGMOD*, 2010.

[9] L. Barbosa, D. Caseiro, G. Di Fabbrizio, and A. Stent, “SpeechForms: From web to speech and back,” in *Proceedings of INTERSPEECH*, 2011.

[10] S. Issar, “A speech interface for forms on WWW,” in *Proceedings of the European Conference on Speech Communication and Technology*, 1997, pp. 1343–1346.

[11] D. Bobrow and B. Fraser, “An augmented state transition network analysis procedure,” in *Proceedings of IJCAI*, 1969.

[12] D. Traum and S. Larsson, “The information state approach to dialogue management,” in *Current and New Directions in Discourse and Dialogue*, J. van Kuppevelt and R. Smith, Eds. Springer, 2003.

[13] B. Liu, R. Grossman, and Y. Zhai, “Mining data records in web pages,” in *Proceedings of KDD*, 2003.

[14] Y. Zhai and B. Liu, “Web data extraction based on partial tree alignment,” in *Proceedings of WWW*, 2005.

[15] W. Liu, X. Meng, and W. Meng, “Vide: A vision-based approach for deep web data extraction,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 3, pp. 447–460, 2010.