A DATA-CENTRIC ARCHITECTURE FOR DATA-DRIVEN SPOKEN DIALOG SYSTEMS*

Sebastian Varges, Giuseppe Riccardi

Department of Information and Communication Technology
University of Trento
38050 Povo di Trento, Italy

ABSTRACT

Data is becoming increasingly crucial for training and (self-) evaluation of spoken dialog systems (SDS). Data is used to train models (e.g. acoustic models) and is 'forgotten'. Data is generated on-line from the different components of the SDS system, e.g. the dialog manager, as well as from the world it is interacting with (e.g. news streams, ambient sensors etc.). The data is used to evaluate and analyze conversational systems both on-line and off-line. We need to be able query such heterogeneous data for further processing. In this paper we present an approach with two novel components: first, an architecture for SDSs that takes a data-centric view, ensuring persistency and consistency of data as it is generated. The architecture is centered around a database that stores dialog data beyond the lifetime of individual dialog sessions, facilitating dialog mining, annotation, and logging. Second, we take advantage of the state-fullness of the data-centric architecture by means of a lightweight, reactive and inference-based dialog manager that itself is stateless. The feasibility of our approach has been validated within a prototype of a phone-based university help-desk application. We detail SDS architecture and dialog management, model, and data representation.

Index Terms— Spoken Dialog Systems, Dialog System Architecture, Dialog Management, Data Management, VXML Generation

1. INTRODUCTION

Data is becoming increasingly important for spoken dialog systems, not just in speech processing but also in spoken language understanding, dialog management and language generation. It is used for data mining, training of language models, system evaluations, and forms the basis for annotation efforts. In contrast to previous work that treats data as peripheral and focuses on inter-agent communication [1, 2], in our approach a database management system (DBMS) is at the center of our system, both handling data and managing communication flow. Database triggers and server-side stored procedures make the architecture reactive, resulting in a *pervasive blackboard model* for dialog management, spoken language understanding and generation (in contrast to standard *pipeline* architectures). From a practical point of view, the availability of high-quality open source DBMSs such as MySQL and PostgreSQL, and of free versions of commercial DBMSs such as DB2, Oracle and MS SQL Server, allows one to take advantage of the robustness and scalability of these systems.

We distinguish *dialog management* from the *architecture* level. A dialog manager (DM) keeps track of dialog moves and turn taking between participants, and generally maintains the dialog context [3, 4, 5, 6, 7]. Within the DM, particular *dialog models* can be realized, for example based on the notions of state machine, ATN/RTNs, or Information State Update. In this paper, we present a lightweight, reactive, and inference-based dialog manager as an example of the kinds of DMs afforded by the architecture. The DM takes advantage of the state-fullness of the data-centric architecture by being itself stateless, i.e. purely functional. Its input is selected from the database by SQL statements, and its output is again stored in tables in the database. The output of the dialog manager is a set of declarative statements that are used to dynamically generate Voice XML (VXML; [8]) pages. Our dialog model is a combination of state-based and Information State Update approach: the dialog manager moves from named state to named state by making transitions that are conditioned on a broader Information State (dialog context). Once a new named state is reached, this context is updated.

The feasibility of our approach has been validated in a prototype of a phone-based help-desk application at our University. Its functionality has been modeled after a previously and independently developed pure VXML/PHP application. In this paper, we use examples from this application to illustrate dialog management, model, and data representation.

This papers is organized as follows: in section 2 we describe the data-centric architecture, including possible usage scenarios (sec. 2.1), relevant database characteristics and general SDS architecture (sec. 2.2), options for implementing processing modules (sec. 2.3), the communication protocols for the VXML server and basic data organization for SDS (sec. 2.4), and related work on SDS architectures (sec. 2.5).

*THIS WORK WAS PARTIALLY SUPPORTED BY THE EUROPEAN COMMISSION MARIE CURIE EXCELLENCE GRANT FOR THE ADAMACH PROJECT (CONTRACT NO. 022593).

In section 3, we discuss the dialog manager: its general properties within the architecture (sec.3.1), the characteristics of our particular dialog manager (sec.3.2), the implemented dialog model for the help-desk application (sec.3.3, 3.4), and related work (sec.3.5). Section 4 concludes this paper.

2. A DATA-CENTRIC ARCHITECTURE FOR SDS

2.1. Data-generating scenarios

We envision various types of data and data-generating scenarios, which we want a SDS architecture to be able to handle. First, data is generated during user interaction, either by human users (in which case maintaining a *user profile* across sessions will become increasingly important), or by Wizard-of-Oz experiments, or by simulated users (e.g. for policy planning in reinforcement learning [9, 10]). Second, annotation is a source of data, either in the form of off-line annotation or as annotation during system interaction (for example, a human supervisor commenting on the actions of the SDS). Additional training data produced in other contexts/projects will often be used. In all the above cases, data may also be multi-modal (gesture, face recognition, eye tracking etc). There may be multiple speakers in the same dialog, or multiple dialogs running independently (where one would want to learn from one user and apply the results to next one, for example).

Separate from the dialog aspect, domain/task data, frequently non-linguistic, may arrive asynchronously, originating from web mining, sensor data streams, or queries to structured domain data (e.g., flight schedules). These examples show that we will need to handle large amounts of heterogeneous data, provide asynchronous read/write access and a communication infrastructure for the various SDS components. We thus need a flexible architecture, without constraining future developments.

2.2. Database characteristics and SDS architecture

Our proposal is to use a Database Management System as the central component of the SDS. Current DBMSs offer capabilities in two dimensions: First, a theoretically well-defined data model, i.e. a model of data and data manipulation (SQL), which is becoming increasingly hybrid, combining relational and XML-based data (with extensions to the SQL query language). Second, DBMSs provide an implementation of a database server that allows one to store terabytes of data and includes mechanisms for replication and concurrent access and communication, amongst others.

Figure 1 shows a sketch of the overall architecture, with the DBMS at the heart of the system, taking into account some of the scenarios outlined above. The user accesses the VXML platform by means of either a fixed phone line or an IP SoftPhone. In our implementation, a PostgreSQL database receives 'speech events' from a VXML speech server and

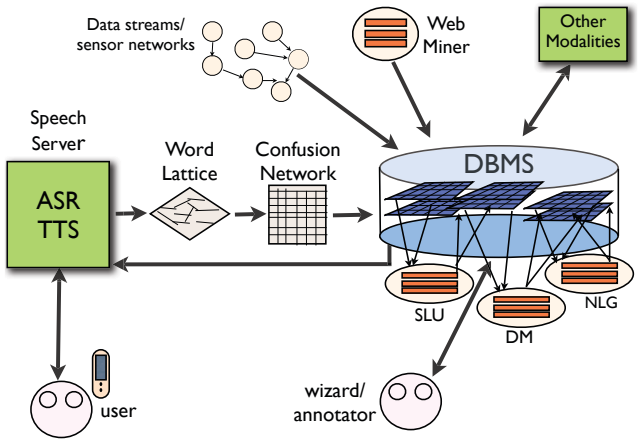

Fig. 1. Vision of general architecture for data-centric SDS

makes them available to a dialog manager (see section 2.3), which in turn posts its response to the database, where it is picked up by the VXML server for TTS. However, a data-(base)-centric architecture is more general than our specific implementation: it can handle a wide range of data, from multi-modal input, various content sources to human (wizard) intervention in the dialog. Furthermore, ASR input could be provided by SLMs and stored as word lattices or word confusion networks in the database.

2.3. Processing modules

DBMSs provide two mechanisms that are relevant for its use within a dialog system: *triggers* and *stored procedures*. Triggers initiate a trigger function when an SQL-relevant event occurs (INSERT, UPDATE, or DELETE statements). Thus, Triggers make a system *event-driven* and enable a blackboard-style architecture that, for example, reacts to 'speech events'. Trigger functions can be stored procedures, which are user-defined, server-side functions that are loaded upon first use. They can access the database tables and can be written in a variety of programming languages, depending on the DBMS.

The combination of triggers and stored procedures results in a wide range of implementation options for processing modules such as dialog management and language understanding and generation: *all* modules of the SDS could be defined as stored procedures, which has the advantage of built-in concurrency, speed and avoidance of module/data interface problems. However, it also requires one to use the implementation languages that are available in the DBMS, and there may be restrictions on the use of stored procedures. A more conventional option is to use triggers to call modules/agents that live outside the DBMS. This option can benefit from the client libraries that are usually available for DBMSs. However, it also requires a greater effort for developing independent processing agents. Here, we explore a third option that combines the two extremes: we use state-less functions similar to stored

procedures but implement them outside the DBMS (by using triggers that call dialog management functions at the OS-level). This has the advantage that any function that can be invoked on the OS-level can be called from the DBMS, giving us much greater flexibility of implementation choices than stored procedures inside the DBMS. Data exchange with the DBMS is done by invoking standard, safe data access functions.

2.4. Dialog Data Modeling

Determining the appropriate table structure is an important step in designing a data-centric SDS, since all dialog data is represented in the database. This includes questions such as where triggers should be attached, how primary keys should be defined (taking into account that many dialog sessions will be stored), and what queries we want to ask.

We distinguish between three sets of tables: the basic stream of utterances of the participants, the dialog manager state, and the user/task model (the latter two are described in section 3.3). The basic stream of utterances minimally comprises user and system utterances. They are represented by separate tables, which one can think of as parallel 'tracks' since participants may speak concurrently (as a consequence, each table has independent 'turn ids'). The precise table structure constitutes the *communication protocol* to/from the speech server, and depends on the information provided by ASR and accepted by TTS.

The nbest ASR results of the VXML platform are stored in a table `user_turns`, including utterance, VXML interpretation, rank, confidence, session-ids, and timestamps. ASR-relevant information is collected by an ECMAScript [11] function and inserted into the database by PHP scripts. There are separate tables that record VXML events such as <nomatch> and <noinput>, which do not result in new rows in table `user_turns`.

System turns generated by the dialog manager follow the communication protocol from DBMS/DM to VXML server. They contain system utterance, turn-id and various parameters for dynamic VXML generation, e.g. `maxnbest`, `confidencelevel`, `sensitivity`, grammar name for recognition and languages to be used for TTS and ASR. The dialog manager can produce a set of ranked responses (section 3.2); only the highest ranked system turn is verbalized.

Since all dialog data is stored in the database, we can obtain the utterances of the participants of an ongoing dialog by a SQL UNION of the relevant rows of the two tables `user_turns` and `system_turns` ordered by time (table 1).

2.5. Related work

Current dialog system architectures such as the Open Agent Architecture [1] or DARPA Communicator [2] provide a (centralized) communication infrastructure that matches information providers with recipients. The set of triggers in our approach serves a similar role to Communicator hub scripts. However, current approaches treat data as peripheral: they were not designed to handle large volumes of data, and data is not automatically persistent. On the other hand, considerable effort has been spent on developing external logging facilities, for example [12].

The Florence dialog manager/architecture [3] also takes advantage of industry standards, albeit in the different area of web application servers. The Florence DM inherits the session management capability of these servers to serve multiple users. However, dialog data is not automatically persistent across sessions or is accessible to all its modules (pipeline model).

Pure VXML platforms severely limit the choice of dialog modeling options, processing modules, and suffer from a similar lack of data persistency. However, they can provide the basis for other systems, as we have done in this work.

3. DIALOG MANAGEMENT

3.1. Functional dialog management

As as consequence of the stateful, data-centric architecture, the dialog manager, i.e. the component that encapsulates the 'dialog logic' by analyzing user utterances and generating system turns, can be stateless. (As outlined above, there are alternative options for more conventional, stateful dialog management.) A stateless dialog manager can be defined as a function from sets of database n-tuples (the domain d of the function) to sets of database tuples (its range r):

$$f_{DM}(r_{d1}, r_{d2}, ..., r_{dn}) \longrightarrow (r_{r1}, r_{r2}, ..., r_{rn}) \qquad (1)$$

where each r_{xn} represents a (possibly empty) set of n-tuples of an n-ary *relation*. In our implementation, the functional DM reads from and writes to the same relations, i.e. tables. Function f_{DM} is a *partial* function since the n-tuples in the domain of the function form a subset of the set of n-tuples of the relation.

A functional approach to dialog management inherits the advantages of functional computational models in general: functions can be parallelized, which in this context means that the DM can entertain separate dialogs with an in principle arbitrary number of users at the same time (in between invocations to respond to user A, it can respond to user B, etc). Furthermore, since the result of a function call only depends on its inputs, functions calls can be repeated, which in practice facilitates development and the construction of test cases. Any DM state can be retrieved from the database, which means that states deep into dialog can just as easily be reinvoked as states at the beginning of the dialog, without the need to run the dialog system through a sequence of user turns from the 'start state'.

turn_id	participant	utterance
3913	system	Welcome to the University Help Desk! You can either subscribe to an exam , or verify an exam , or cancel a registration . How may I help you?
1453	user	subscribe
3914	system	I see, you have chosen subscribe . Please tell me the the name of the exam.
1454	user	informatics one
3915	system	Okay, please tell me your student I_D number, spelling each digit individually .
1455	user	one two oh three four five
3916	system	I understood student I D 1 2 0 3 4 5 , is that correct?
1456	user	no no
3917	system	Please tell me again your student I_D number.
1457	user	one two three four five
3918	system	I understood student I D 1 2 3 4 5 , is that correct?
1458	user	yes correct
3919	system	Do you confirm that you would like to subscribe to exam Informatics One, and that you have student I_D number 1 2 3 4 5 ?
1459	user	yes
3920	system	Thank you, your data has been stored. Good Bye.

Table 1. Result of SQL UNION of (subset of) tables user_turns and system_turns

An apparent disadvantage of functional dialog management is the need to reinvoke the dialog manager for every new input, while the user is waiting for a system response. However, we argue that first, the DM can be lightweight due to the data-centric architecture, and second, it is often possible to speed up start-up times even for interpreted (rather than compiled) implementation languages by saving 'images' – this is what we do in our system; see below. Furthermore, this issue does not arise at all for dialog managers implemented as stored procedures inside the DBMS since these are loaded only once upon first use.

It is an important aspect of a functional approach to dialog management that the DM can be supplied with relevant information such as the user profile *before* it processes the next turn: in the implementation described in sections 3.2 and 3.3, we always execute a SQL query that tries to obtain the relevant user profile; as long as the user's ID is not known (`'nil'`), the result set of the query is empty, and the DM has to generate its response without detailed user profile. (Of course, one could alternatively issue a separate database query during DM processing, but this is not necessary.)

3.2. An inference-based dialog manager

Our dialog manager, in addition to being functional, is *inference-based*: it uses a set of inference rules/productions to transform input data (ASR results, the current DM state, and relevant domain knowledge) into output data (system responses, DM states, and domain knowledge). Inference rules *fire* by executing a function when a set of conditions is met. These conditions react to changes in the program-internal database (the *knowledge base*, KB), resulting in a blackboard-style control flow [13]. Thus, our approach uses two blackboards, a coarse-grained one at the architecture level (to activate modules such as SLU or DM), and a fine-grained one at the dialog management level. An inference-based dialog manager allows one to easily integrate task-level inferences (next section), and it is adaptive in the sense that new rules can be constructed on-the-fly (not exploited yet).

The inference-based DM can produce more than one response and result state, each derived from another ASR hypothesis: inference rules fire for every matching set of facts and do not need to be limited to ASR hypotheses of rank 1. Since the database can be used to store large numbers of states, this opens up a way to model probability distributions over possible states [9, 10]. In our current implementation, we store different states and system responses in the database, but only retrieve the highest-ranked state (based on ASR confidence) for the next system turn. Thus, we effectively do search with a beam size of 1.

3.3. Dialog model and example application

Our *dialog model* can be characterized as a hybrid of state machine and Information State approach [5]: the system follows transitions from named states to named states. We thus distinguish two notions of dialog 'state': the Information State (IS), which is a subset of the KB, and the 'named state' in the sense of a state machine, which consists of just one type the fact. These transitions are conditioned on further information that needs to be present in the IS, without the need to match the entire IS. For example, the rule in figure 2 applies if the system was 'in' state 5 in the previous turn[1], the VXML interpreter found a grammar match for the user utterance (`'vxml_event filled'`), which can be interpreted

[1]For simplicity, we omit the book-keeping of turn-ids etc: state 5 is the state of the previous turn.

```
(defrule user-answer-possible-student-id
  (state (name 5))                                                                    ; condition 1
  (control-asr (vxml_state dialog_loop) (vxml_event filled))                          ; condition 2
  (user-turn (interpretation ?student-id) (rank ?rank)                                ; condition 3
             (confidence ?confidence&:(< ?confidence (confidence-threshold student-id))))
  (possible-student-id ?student-id)                                                   ; condition 4
 =>
  (assert (system-response (utterance (generate-yesno-question student-id ?student-id))   ; action 1
   (application exam) (rank ?rank)))
  (assert (move-yesno-question  (participant system)  (parameter  student-id) (value ?student-id) ; action 2
                                (value-confidence ?confidence) (value-rank ?rank)
                                (next-positive 8) (next-negative 9)))
  (assert (state (name 7))))                                                           ; action 3
```

Fig. 2. Rule that expects user answer providing student ID

as a possible student ID. This is tested by requiring the presence of a fact 'possible-student-id' for the ID, which is generated by domain inferences triggered by the corresponding ASR result. This shows the tight integration of dialog processing with domain-level reasoning that is possible in an inference-based DM: a 'possible ID' is a sequence of a certain number of digits; there is the further notion of a 'valid ID' which is based on the actual student IDs in the database. It can be used in the next system turn if the user confirms the yesno question produced by the rule in figure 2, which happens if the confidence of the ASR result is below a given threshold (which can change dynamically since confidence-threshold is a function call). More precisely, the right-hand side of the rule generates three facts: First, a system response that is used for TTS. It contains a large number of default values (not shown) to comply with the DB-to-TTS communication protocol (for TTS language, barge-in etc; section 2.4). Second, a move-yesno-question is generated that states the parameter in question and its value, and the next named states the system should visit in case the user gives a positive or negative answer (next-pos; nextneg). This fact is stored in the database and retrieved for the next system turn, i.e. it becomes part of the Information State. Finally, the rule generates a new named state 7.

One of rules that can be used in the next system turn (not shown) expects the DM to be 'in' state 7, a yesno question of the previous turn, and a "yes" answer by the user that is ranked more highly than a "no" answer. If these conditions are met, the parameter in question is added as explicitly confirmed to the application information for the current session. Like the dialog context, the application parameters are stored in the database (table 2).

Once the user has supplied all the required application parameters and confirms them, the chosen action is stored in a separate table for exam subscriptions, which is the final outcome of the dialog. It is only possible to cancel exams for which the user has subscribed to before (an appropriate response is generated if cancellation fails).

The currently implemented dialog manager uses 30 rules for modeling dialog with 15 named states. It furthermore contains 31 generation functions that generate system responses. The language of the responses (and the ASR/TTS settings) can be switched between English (US and GB), and Italian and German.

user_turn_id	parameter	value	confd.	confirmation
1453	goal	register	0.562	assumed
1454	exam-name	info_1	0.839	assumed
1457	student-id	12345	0.774	YES_explicit

Table 2. Application parameters in DB for dialog in table 1

3.4. Preliminary SDS prototype

We tested a prototype of our SDS from the point of view of real-time processing by asking 6 users (colleagues), who were given student IDs in advance, to call the system and subscribe to an exam (in Italian). They were then asked to call again and cancel the subscription. (Some were also asked to try and cancel a non-existing subscription or exam.) 5 users accomplished the task, with 2 restarts/calls required. The use of a database allows us to analyze the user dialogs directly by using SQL queries. We find that the mean response time of the SDS architecture is 849 msecs ($\sigma = 46$ msecs), as measured by the time difference between ASR entries (entered by the VXML server into the database) and system responses (entered by the DM). We are currently evaluating the system further and will report results at the workshop.

3.5. Discussion and related work

It is instructive to compare the inference-based approach with a pure VXML application: A VXML <form> element typically encapsulates a system prompt, a grammar specification for matching the user response, some case distinctions for the different responses/values, and VXML event handlers. In an inference-based approach, each of these are handled by different rules; in particular, rules do not cross 'turn boundaries'.

In contrast to a pure finite state machine, our dialog model is much more compact, using, for example, boolean expres-

sions in rule conditions (`((or (state (name 7)) (state (name 6)))`), or variable state names, which we use to handle VXML events generically.

Our dialog *model* is similar to an ATN, which can be implemented in an Information State-based dialog *manager* [5]. There is clearly a trade-off between the compactness of named states, and the generality of explicit representations of dialog context: it seems that named states are more meaningful for a system-initiative dialog model than for a user-initiative one.

Our inference-based DM is similar to the Dipper [6] and TrindiKit [5] DMs in its use of condition-action rules. A key difference is that our architecture handles data management and communication, increasing robustness and allowing the DM to be more lightweight. Furthermore, our DM uses a standard production system and therefore inherits the ability to efficiently perform matching with complex conditions and large sets of rules and facts.

4. CONCLUSIONS

We propose a data-centric architecture that is designed to meet the future needs of data-intensive processing of heterogeneous types of data. By using a standard database management system, we take advantage of available industrial strength infrastructure and support tools (JDBC/ODBC database drivers, client libraries, GUI DB management tools etc.). The presented data-centric system uses a purely relational data model. This could be combined in the future with XML data which is becoming available in hybrid relational-XML DBMSs.

We take advantage of the architecture by defining an inference-based dialog manager that is purely functional, and realizes a hybrid finite state/Information State-based dialog model. A functional DM enforces a strict discipline of storing all relevant data in the database. The architecture can also be used with stateful processing modules. The combination of a relational database with a production system for dialog management largely eliminates the 'impedance mismatch' that is typical for example for object-relational systems: the data structures (facts) of the DM directly correspond to rows in the database.

5. REFERENCES

[1] David L. Martin, A. J. Cheyer, and D. B. Moran, "The Open Agent Architecture: A framework for building distributed software systems," *Applied Artificial Intelligence: An International Journal*, vol. 13, no. 1-2, pp. 91–128, January-March 1999.

[2] Stephanie Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, "GALAXY-II: A reference architecture for conversational system development," in *Proc. of ICSLP 1998*, Sydney, Australia, 1998.

[3] Giuseppe Di Fabbrizio and Charles Lewis, "Florence: a dialogue manager framework for spoken dialogue systems," in *Proc. of INTERSPEECH '04*, Jeju Island, Korea, 2004.

[4] Roberto Pieraccini, Sasha Caskey, Krishna Dayanidhi, Bob Carpenter, and Michael Phillips, "ETUDE: A recursive dialog manager with embedded user interface patterns," in *Automatic Speech Recognition and Understanding Conference 2001 (ASRU)*, Keystone, Colorado, 2001.

[5] Staffan Larsson and David Traum, "Information State and dialogue management in the TRINDI Dialogue Move Engine Toolkit," *Natural Language Engineering*, vol. 6, no. 3–4, pp. 323–340, 2000.

[6] Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka, "DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture," in *SIGdial Workshop on Discourse and Dialogue*, Sapporo, Japan, 2003.

[7] Danilo Mirkovic and Lawrence Cavedon, "Practical Plug-and-Play Dialogue Management.," in *Proceedings of the 6th Meeting of the Pacific Association for Computational Linguistics (PACLING)*, Tokyo, Japan, 2005.

[8] World Wide Web Consortium, "Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Working Draft, 23," See http://www.w3.org/TR/2001/WD-voicexml20-20011023/, October 2001.

[9] Nicholas Roy, Joelle Pineau, and Sebastian Thrun, "Spoken dialog management for robots," in *Proc. Association for Computational Linguistics (ACL-00)*, Hong Kong, 2000.

[10] Jason D. Williams and Steve Young, "Partially Observable Markov Decision Processes for Spoken Dialog Systems," *Computer Speech and Language*, vol. 21, no. 2, pp. 393–422, 2006.

[11] ECMA, "Standard ECMA-262. ECMAScript Language Specification. 3rd Edition," http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM, 1999.

[12] Stephanie Seneff, R. Lau, and J. Polifroni, "Organization, Communication, and Control in the Galaxy-II Conversational System," in *Proc. of Sixth European Conference on Speech Communication and Technology (EUROSPEECH'99)*, Budapest, Hungary, 1999.

[13] Peter Jackson, *Introduction to Expert Systems*, Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.