Ambiguity Representation and Resolution in Spoken Dialogue Systems

Egbert Ammicht, Alexandros Potamianos, Eric Fosler-Lussier

Bell Labs, Lucent Technologies, New Jersey (USA)
{eammicht,potam,fosler}@lucent.com

Abstract

Spoken natural language often contains ambiguities that must be addressed by a spoken dialogue system. In this work, we present the internal semantic representation and resolution strategy of a dialogue system designed to understand ambiguous input. These mechanisms are domain independent; task-specific knowledge is represented in parameterizable data structures. Speech input is processed through the speech recognizer, parser, interpreter, context tracker, pragmatic analyzer and pragmatic scorer. The context tracker combines dialogue context and parser output to yield raw attribute-value (AV) pairs from which candidate values are derived. The pragmatic analyzer adjusts the confidence associated with each AV candidate based on system intent, e.g., implicit confirmation and user input. Pragmatic confidence scores are introduced to measure the dialogue managers confidence for each AV: MYCIN-like scoring is used to merge multiple information sources. Pragmatic analysis and scoring is combined with explicit error correction capabilities to achieve efficient ambiguity resolution. The proposed strategies greatly improve dialogue interaction, eliminating about half of the errors in dialogues from a travel reservation task.

1. Introduction

In natural spoken dialogue interfaces for information retrieval applications, the understanding component of the system must be able to integrate various sources of information to produce a coherent picture of the transaction with the user. Some of this information, however, can be ambiguous in nature. The semantic content of some phrases can be ill defined: "I want to fly *next Saturday*" could mean Saturday of this week or next; "Leave at *six o'clock*" is interpretable as either 6 a.m. or 6 p.m. Furthermore, since speech recognition error rates for natural spoken dialogues are currently relatively high,[1] mistakes made early in the processing chain can propagate throughout the system. Correction of such errors by the user introduces another form of ambiguity, especially since the error correction might itself be in error! Finally, a system must cope with the fact that users might explicitly change their minds, creating a third type of ambiguity.

In order to handle these different sources of ambiguity, the system designer must implement data structures and algorithms to efficiently categorize incoming information. One can, of course, construct *ad hoc* structures to hold ambiguous information (e.g., a specialized "date" class designed to disambiguate phrases such as "next Saturday"). However, our goal is to characterize semantic ambiguity in a domain-independent fashion. In our system, we develop a parameterizable data structure (called the prototype tree) from the ontology of the domain, and define all other operations based on this structure. While not all knowledge about a domain is encodable within the tree, this succinct encapsulation of domain knowledge allows generalized, domain independent tree operations, while relegating the non-encodable specialized domain knowledge into a small set of task-dependent procedures.[2]

In this work, we explore a novel semantic representation and ambiguity resolution system. The system takes in spoken or typed natural language text, and derives candidate attribute-value (AV) pairs (e.g., "Fly to Atlanta in the morning" produces <TOCITY>=Atlanta and <TIME>=morning). We model two types of ambiguities in the system: *value ambiguities*, where the system is unsure of the value for a particular attribute, and *position ambiguities*, where the attribute corresponding to a particular value is ambiguous. We score candidate values based on supporting evidence for or against the candidate. This evidence is provided by raw data, by pragmatic analysis and by matching a hypothesis to the current context. We also provide an override capability to the user based on a dialogue strategy of extensive implicit and explicit confirmation.

While the current application of our system is an automated travel agent capable of handling flight, car rental, and hotel arrangements (the DARPA Communicator task [4]), the algorithms described here are independent of the domain.

2. Semantic representation of ambiguity

Three hierarchical data structures are introduced for representing and instantiating domain semantics. The prototype tree represents the domain ontology, the notepad tree holds all raw values elicited or inferred from user input and the application tree holds the derived candidate attribute values.

2.1. The prototype tree

The basic data structure for representing values is a tree that expresses *is-a* or *has-a* relationships. The tree is constructed from the ontology of the domain: nodes encode concepts and edges encode relationships between concepts [2, 6]. A trip, for example, comprises flights, hotels and cars. A flight, in turn, consists of legs, with departure and arrival dates, times and airports. Data are defined in terms of the path from the tree root to a leaf (the attribute), and the associated value. These paths can be uniquely expressed as a string consisting of a sequence of concatenated node names starting from the root, with a suitable separator such as a period. The attribute for the departure city *Atlanta* shown in Figure 1, for example, is given by ".trip.flight.leg1.departure.city." This tree representation of the semantics – referred to as the *prototype tree* – is domain independent [4].

The prototype tree is an over-specification of the domain; not every concept in the tree will be explored during an inter-

[1] For example, the speech recognition word error rate in the June 2000 DARPA Communicator evaluation was between 20-50%, depending on the system [9].

[2] As an example, when setting up a system for flight information, the fact that a flight has a departure time is encodable within the tree. The fact that a round trip has the same departure city in the first leg as the arrival city in the second leg, however, is only encodable as a task-specific *constraint* on the tree.

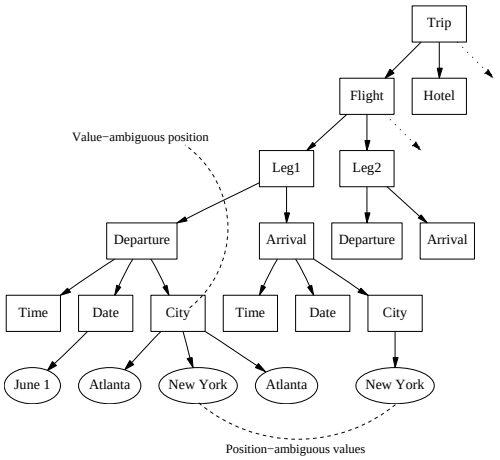

Figure 1: Notepad tree illustrating value ambiguity (departure city Atlanta or New York) and position ambiguity (New York).

action with the user. Information about the types of values a concept can take is also included in the prototype tree. Not all concepts have values (usually non-leaf nodes in the tree such as "departure"). Concepts that take values are referred to as *attributes*. Certain attributes can take multiple values, e.g., airline preferences, while others have to be unambiguously instantiated, e.g., departure city. Some attributes can be instantiated with a constraint rather than a value, e.g., arrival time = "before 5 p.m.". In addition to the semantic information, the prototype tree is overloaded with task and interface information such as the importance of each attribute for task completion. (See [4] for further details.) More complex relationships between concepts, such are timeline consistency checking or inference about the values of an attributes are currently not encoded in the prototype tree and are handled by a separate semantic module.

2.2. The notepad tree

During a dialogue, the system extracts data values from user utterances and places them in the *notepad tree*, a tree structure that mirrors the prototype tree. Values retrieved from the user usually are not accompanied by complete references to the attribute (e.g., "I want to leave from Atlanta" yields a "departure.city" "Atlanta"). The system typically has a partial attribute, e.g., ".trip.flight.leg1", that needs to be merged with "departure.city" to form a complete attribute ".trip.flight.leg1.departure.city". The required context-tracking algorithms are further described in Section 3.

2.3. Ambiguities

Over several dialogue turns, AV pairs may be collected that share the same attribute. For example, we may collect two different departure city names, thereby introducing a value ambiguity. The cause, as discussed in the introduction, may be speech recognizer errors, parsing errors or ambiguous user language. Another possibility is an intentional change of the value by the user. A second kind of ambiguity occurs when the context does not uniquely identify the attribute of a value. For example, we may collect a city, but be unsure whether to classify it as a departure or an arrival city for a given flight, thereby introducing a position ambiguity. Such data will be entered in each of the corresponding positions in the notepad tree as illustrated in Figure 1. The notepad tree structure by itself cannot be used to keep track of position ambiguities, but must be augmented by an additional data structure that indexes position ambiguous nodes.

2.4. The application tree

In most instances, we require that the value of a given attribute be unique. Given a set of value ambiguous entries in the notepad tree, an initial formulation would be to introduce equivalence classes for the data, combined with some disambiguation strategy to select a particular class. We would then be able to use an instance in the class for the desired value. Referring to Figure 1, for example, we could select "Atlanta" as the departure city of the first leg. This approach rapidly proves untenable, however, since, for example, time specifications such as "in the morning" or "after 10 a.m." do not have a useful transitivity relationship.

For the purposes of the application, it may be necessary to expand or to merge various values, e.g., merging times into appropriate time intervals, or expanding cities to a set of airports. Such derived values must be carefully chosen: distinct candidates should lead to different travel itineraries, while still being recognizable to the user as a direct consequence of some user input. Each of the resulting values becomes a distinct candidate for the unique value that we require. Candidates are maintained in a separate data structure, the *application tree*, similar to the notepad tree used to hold the corresponding raw data.

This clear distinction between the raw data and the derived candidates greatly simplifies the formulation of the algorithms, and in particular the selection methods based on the scoring algorithms described below. They will require a suitable definition of consistency between candidate and raw data values.

In order to fill the notepad tree, raw data (either from natural language input or spoken input) is parsed using a recursive finite-state parser [5] that acts as a semantic island parser; it iteratively builds up semantic phrase structures by transducing string patterns in the input into their semantic concepts (e.g., "Atlanta" is rewritten as <CITY>; "arriving in <CITY>" is subsequently transformed into <TOCITY>). The output of the parser is a set of tree branches (islands), which are then transformed, by a set of application-dependent routines in an interpreter, yielding a canonical form the notepad can use. For example, in the travel domain, cities are transformed into airport codes, date strings (e.g., "Tuesday") are converted to the relevant date, and phrases like "first class" are changed into corresponding fare codes.

After raw data is processed by the parser and interpreter, it is placed in the notepad tree, and new candidate values, if any, are derived and placed in the application tree. Note that the formation of suitable candidates and the definition of the associated consistency relationship may be specific to each particular attribute, and hence application dependent. In our current system, position ambiguous data are not used to generate candidates. Instead, they result in the modification of the scores of existing candidates, and are made the subject of clarification dialogues where required.

3. Context tracking

The system maintains an expected context for every user response. This context is expressed as a path r from the root to some node of the prototype tree. Values extracted from a user utterance are in general associated with a partial attribute (i.e., a path l from some prototype tree node to a leaf). Derivation of the correct attribute for a given value requires matching the context to the partial attribute to form a complete path a from the root of the tree to the leaf. In the following discussion we use the operator $\cdot$ to express concatenation.

In general, we need to consider three types of matches: i) **exact match:** the paths match up perfectly, i.e., $a = r \cdot l$ ex-

ists in the prototype tree. For example, given the context r ".trip.flight.leg1" and a datum with partial attribute l ".departure.city", the complete path ".trip.flight.leg1.departure.city" is seen to exist in the tree in Figure 1. ii) **interpolated match:** the path must be interpolated, i.e., for some paths m, the attribute $a = r \cdot m \cdot l$ exists in the prototype tree. For example, a context ".trip.flight" and a datum with partial attribute "departure.city" may be completed with the choice m = ".leg1". iii) **overlap match:** the paths overlap, i.e., there exists a path m such that $r = r' \cdot m$ for which $a = r' \cdot l$ exists in the prototype tree. In this latter case, the general strategy chosen is to minimize the length of the path m. The overlapped substrings may not necessarily have to agree. For example, the context ".trip.flight.leg1.departure" may have to be shortened to ".trip.flight.leg1" so as to combine with ".arrival.city" to form a possible attribute. By shortening the context, we essentially allow the user to override the system context.

An interpolation match can lead to position ambiguities; to control the generation of paths, we introduce the notion of extending a partial attribute l with a given path l' prior to attempting a match. To do so, we specify tuples of the form (l, l', r', γ) that are derived from the prototype tree and form a parameterization of the following algorithm: given a context of the form $r = r'm$ and a partial attribute l, use any of the other matching algorithms applied to r and the extended path $l' \cdot l$. If successful, the parameter γ in $(0, 1]$ is used as a weight in the scoring algorithm (see Eq. 2). This mechanism allows us to exclude undesirable paths such as a "stopover.city", or to select one path over another. An example of the latter would be to favor an "arrival.date" for ".trip.cars", i.e., the date when the car will be picked up, over a ".trip.cars.departure.date", while favoring ".departure.date" for ".trip.flight", i.e., the date of departure for a flight.

A further refinement of the context-tracking system is to allow for context changes while analyzing data from a given user utterance. These changes can be made unconditionally, or may be pushed on a stack, with previous contexts searched if the current context should fail within the current utterance.

4. Scoring mechanism

Given multiple candidates for a given attribute, we require a sufficiently parameterized value selection mechanism amenable to training, such that a reasonable dialogue will result. To this effect, we assign scores, i.e., MYCIN style [8, 1] confidence factors s with range [-1, 1] to every candidate, and define two parameters σ_1 and σ_2 to govern their selection. For a candidate value to be considered, its score must be sufficiently large, $s \geq \sigma_1 \geq 0$. If more than one candidate is present, we further require that the score difference Δs of the top two candidates be sufficiently large, $\Delta s \geq \sigma_2 \geq 0$, for the top candidate to be selected. We currently trained the system to use the settings σ_1=.25, σ_2=.25.

The scores are modified by evidence e for or against the individual candidate value by

$$S(s,e) = \begin{cases} s + (1-s)e, & s \geq 0, e \geq 0 \\ s + (1+s)e, & s < 0, e < 0 \\ \frac{s+e}{1-\min(|s|,|e|)}, & \text{otherwise,} \end{cases} \quad (1)$$

Such evidence arises from individual data in the corresponding notepad positions, from the match-type and the number of attributes for a value obtained by the context-tracking algorithm, from direct and/or indirect confirmation based on pragmatic analysis of user responses to a given prompt, and from various rules and constraints that are specific to the system.

Notepad data may have any number of scores attached, (e.g., acoustic confidences and distribution frequencies). To use notepad data as evidence, we combine these scores to derive a single individual score p with range $[0, 1]$. The actual combination function depends on the available score types, and the specific attribute of the datum. The strength of the evidence e for datum **p** used in the update for a candidate **s** is obtained from this score by

$$e = \begin{cases} \alpha p, & \text{if } \mathbf{p} \text{ and } \mathbf{s} \text{ are consistent} \\ \beta p, & \text{otherwise.} \end{cases} \quad (2)$$

where $\alpha \in (0, 1]$ and $\beta \in [-1, 0]$ are constants derived from the context tracking algorithm. We currently use $\alpha = \gamma/n, \beta = -.2\gamma/n$, where γ is the weight of the extended partial attribute context-tracking algorithm or 1, and n is the number of position ambiguous attributes found for the datum.

The scoring operation thus proceeds as follows: i) as each datum is added to the notepad, its score is computed; ii) the scores of all known candidates are updated based on this score; iii) new candidates, if any, are produced; iv) their score is computed based on the available raw data. Note that the score of a candidate is independent of the order in which the scores are updated.

Currently, each datum is inserted into the notepad with a default score p. However, in future versions of our system, we plan to implement end-to-end confidence scores [3, 7] that will take into account the word-level confidence that the speech recognition was correct, and the confidence of the interpreter (e.g., "next Friday" is more likely to mean Friday next week rather than Friday this week).

5. Pragmatic analysis

The above scoring machinery results in dialogues that depend on the scores associated with data extracted from user utterances, the careful tuning of the score combination functions and the parameters, and on the number of times a particular datum is found. Thus, the system remains vulnerable to misrecognitions and systematic errors. The user may be forced to specify a particular datum repeatedly to overcome a large positive score for some candidate.

To mitigate this vulnerability, we use both direct and indirect confirmation of individual data. The system carries out a pragmatic analysis of the user response to a particular system prompt. Individual candidates that may be in error are identified, and candidate scores are modified based on the derived evidence. The phrasing and the data presented in a prompt are designed, therefore, so that three predictions can be made about the user response: i) a possible *confirmation* (yes/no) of an explicit question asked by the system, ii) *expected values and attributes* that should appear if the presented data are correct, and iii) *unexpected values and attributes* that may appear if the user objects to one or more of the data that were presented.

Data extracted from the user utterance is analyzed accordingly, and compared to the expectation. In our current system, we consider responses that fully meet the expectation, but do not contain any unexpected values or attributes as evidence for the values being implicitly confirmed. Special cases, such as a "no" response to an explicit value ambiguity disambiguation question, (e.g., "Are you leaving from Atlanta or from New York?") is taken as strong evidence against both values. Once again, system behavior will depend on careful tuning of these system actions. Conservative settings will result in needless dialogue to reinforce the score of particular AV pairs. Aggressive settings allow the system to make unequivocal decisions based on the pragmatics, while making it correspondingly more difficult

Dialogue		System inferences
S:	How can I help you?	leg1.departure.city=Baltimore (score .5)
U:	I am leaving Baltimore	
S:	Leaving Baltimore. What is the arrival city?	User said: <yes>
U:	fine Portland *(ASR error)*	leg1.arrival.city=Portland (score .5)
		leg1.departure.city=Baltimore (score .65) (implicit confirmation)
S:	Arriving in Portland-Oregon.	User said: <no>
	On what date do you want to depart Baltimore?	Change request: city=Portland to city=Frankfurt
U:	about no not Portland-Oregon Frankfurt-Germany	leg1.arrival.city=Portland (score 0.0)
		leg1.arrival.city=Frankfurt (score 1.0)
		pragmatic analysis confirms problem with arrival city
S:	Arriving in Frankfurt.	city=Fort Wayne
	On what date do you want to depart Baltimore?	change context from departure.date $\rightarrow$ departure.city
U:	like Fort-Wayne *(ASR error)*	system detects conflict Baltimore (s=.61) $\neq$ Fort Wayne (s=.44)
S:	Was the departure city Baltimore or Fort Wayne?	. . .

Figure 2: Correction of mistakes currently handled by the system. S=system utterance, U=user utterance from speech recognition.

for the user to correct errors. The settings must thus account for system capabilities, and in particular, for whether correction mechanisms are available to the user.

6. Correction mechanisms

To allow for aggressive system settings, we have provided a capability for the user to talk about attributes directly, and to request specific actions from the system. In particular, we allow **Information requests**, e.g., "what is the departure city?", to ascertain the value for a specific attribute; **Clear requests**, e.g., "clear the departure city", to force the removal of all candidate values for a given attribute; **Freeze requests**, e.g., "freeze the departure city", to inhibit the system from further changing the value of a particular attribute; **Change requests**, e.g., "change Atlanta to New York", "change the departure city to New York", or even "not Atlanta, New York!", implemented as a clear operation followed by creation of a candidate for the given attribute.

The implementation of these features uses the same context-tracking algorithm to derive the required attribute, with the added requirement that the resulting attribute must be unique. These algorithms are application independent.

7. Analysis of Experimental Results

We analyzed 35 dialogues collected during the third week of the June 2000 DARPA Communicator evaluation [9]. This collection consists of interactions with our previous system [4] which did not have pragmatic scoring, pragmatic analysis and correction mechanisms. Every dialogue was run through both the old system and new system; at turns where the transactions diverged,[3] we characterized the system behavior based on knowledge of the system prompts and speech recognizer outputs, together with information about the internal system state. A sample interaction is shown in Figure 2. Given 49 turns where the systems diverged, we found that our current system improved in 25 cases, compared to 3 cases where the older system was superior. The improvements were due to better parsing in 10 cases, the introduction of scoring and pragmatic analysis in 10 cases, and the interaction of both modules in 3 cases. In 21 cases, the dialogue diverged in ways that did not allow such a value judgment to be made.

[3]The dialogues available contained multiple transactions, frequently covering more than one leg of a flight. Since the systems may therefore reconverge, we had a total of 49 dialogue fragments to analyze.

8. Conclusion

Our novel representation of semantic information within a spoken dialogue system vastly improves our previous system by directly representing ambiguities introduced both by system errors and user directions. The key point of this representation is the separation of user input, task specification, recording of candidate values, and candidate selection into separate, but related, data structures. The addition of a pragmatic scoring mechanism, and the ability for the user to directly talk about attributes in correcting mistakes has improved several dialogues that were previously problematic. In future work, we plan to investigate how to further tune the the scoring algorithms and integrate confidence measures from other system modules.

Acknowledgments: This work was partially funded by DARPA under the auspices of the Communicator project. The authors would like to express their sincere appreciation to Chin-Hui Lee, Jeff Kuo and Andy Pargellis for many helpful discussions.

9. References

[1] D. Heckerman, "Probabilistic interpretations for MYCIN's certainty factors," in *Uncertainty in artificial intelligence* (L. Kanal and J. Lemmer, eds.), (Amsterdam: North Holland), pp. 11–22, 1986.

[2] J. Hobbs and R. Moore, *Formal Theories of the Commonsense World*. Norwood, NJ: Ablex, 1985.

[3] K. Komatani and T. Kawahara, "Generating effective confirmation and guidance using two-level confidence measures for dialogue systems," in *ICSLP*, (Beijing, China), Oct. 2000.

[4] A. Potamianos, E. Ammicht, and H.-K. Kuo, "Dialogue management in the Bell Labs communicator system," in *ICSLP*, (Beijing, China), Oct. 2000.

[5] A. Potamianos and H.-K. Kuo, "Speech understanding using finite state transducers," in *ICSLP*, (Beijing, China), Oct. 2000.

[6] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice Hall, 1995.

[7] R. San-Segundo, B. Pellom, K. Hacioglu, W. Ward, and J. Pardo, "Confidence measures for dialogue systems," in *Proc. ICSLP*, (Salt Lake City), May 2001.

[8] E. Shortliffe, *Computer-based medical consultation: MYCIN*. New York, NY: Elsevier, 1976.

[9] M. Walker et al, "DARPA Communicator dialog travel planning systems: the June 2000 evaluation," submitted to *EUROSPEECH*, (Aalborg, Denmark), 2001.