Variant Transduction: A Method for Rapid Development of Interactive Spoken Interfaces

Hiyan Alshawi and **Shona Douglas**

AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932, USA
{hiyan,shona}@research.att.com

Abstract

We describe an approach ("variant transduction") aimed at reducing the effort and skill involved in building spoken language interfaces. Applications are created by specifying a relatively small set of example utterance-action pairs grouped into contexts. No intermediate semantic representations are involved in the specification, and the confirmation requests used in the dialog are constructed automatically. These properties of variant transduction arise from combining techniques for paraphrase generation, classification, and example-matching. We describe how a spoken dialog system is constructed with this approach and also provide some experimental results on varying the number of examples used to build a particular application.

1 Introduction

Developing non-trivial interactive spoken language applications currently requires significant effort, often several person-months. A major part of this effort is aimed at coping with variation in the spoken language input by users. One approach to handling variation is to write a large natural language grammar manually and hope that its coverage is sufficient for multiple applications (Dowding et al., 1994). Another approach is to create a simulation of the intended system (typically with a human in the loop) and then record users interacting with the simulation. The recordings are then transcribed and annotated with semantic information relating to the domain; the transcriptions and annotations can then be used to create a statistical understanding model (Miller et al., 1998) or used as guidance for manual grammar development (Aust et al., 1995).

Building mixed initiative spoken language systems currently usually involves the design of semantic representations specific to the application domain. These representations are used to pass data between the language processing components: understanding, dialog, confirmation generation, and response generation. However, such representations tend to be domain-specific, and this makes it difficult to port to new domains or to use machine learning techniques without extensive hand-labeling of data with the semantic representations. Furthermore, the use of intermediate semantic representations still requires a final transduction step from the intermediate representation to the action format expected by the application back-end (e.g. SQL database query or procedure call).

For situations when the effort and expertise available to build an application is small, the methods mentioned above are impractical, and highly directed dialog systems with little allowance for language variability are constructed.

In this paper, we describe an approach to constructing interactive spoken language applications aimed at alleviating these problems. We first outline the characteristics of the method (section 2) and what needs to be provided by the application builder (sec-

tion 3). In section 4 and section 5 we explain variant expansion and the operation of the system at runtime, and in section 6 we describe how confirmation requests are produced by the system. In section 7 we give some initial experimental results on varying the number of examples used to construct a call-routing application.

2 Characteristics of our approach

The goal of the approach discussed in this paper (which we refer to as "variant transduction") is to avoid the effort and specialized expertise used to build current research prototypes, while allowing more natural spoken input than is handled by spoken dialog systems built using current commercial practice. This led us to adopt the following constraints:

- Applications are constructed using a relatively small number of example inputs (no grammar development or extensive data collection).
- No intermediate semantic representations are needed. Instead, manipulations are performed on word strings and on action strings that are final (back-end) application calls.
- Confirmation queries posed by the system to the user are constructed automatically from the examples, without the use of a separate generation component.
- Dialog control should be simple to specify for simple applications, while allowing the flexibility of delegating this control to another module (e.g. an "intelligent" back-end agent) for more complex applications.

We have constructed two telephone-based applications using this method, an application to access email and a call-routing application. These two applications were chosen to gain experience with the method because they have different usage characteristics and back-end complexity. For the e-mail access system, usage is typically habitual, and the system's mapping of user utterances to back-end actions needs to take into account dynamic aspects of the current email session. For the call-routing application, the back-end calls executed by the system are relatively simple, but users may only encounter the system once, and the system's initial prompt is not intended to constrain the first input spoken by the user.

3 Constructing an application with example-action contexts

An interactive spoken language application constructed with the variant transduction method consists of a set of contexts. Each context provides the mapping between user inputs and application actions that are meaningful in a particular stage of interaction between the user and system. For example the e-mail reader application includes contexts for logging in and for navigating a mail folder.

The actual contexts that are used at runtime are created through a four step process:

1. The application developer specifies (a small number of) triples $\langle e, a, c\rangle$ where e is a natural language string (a typical user input), a is an application action (back-end application API call). For instance, the string *read the message from John* might be paired with the API call `mailAgent.getWithSender("jsmith@att.com")`. The third element of a triple, c, is an expression identifying another (or the same) context, specifically, the context the system will transition to if e is the closest match to the user's input.

2. The set of triples for each context is expanded by the system into a larger set of triples. The additional triples are of the form $\langle v, a', c\rangle$ where v is a "variant" of example e (as explained in section 4 below), and a' is an "adapted" version of the action a.

3. During an actual user session, the set of triples for a context may optionally be expanded further to take into account

the dynamic aspects of a particular session. For example, in the mail access application, the set of names available for recognition is increased to include those present as senders in the user's current mail folder.

4. A speech recognition language model is compiled from the expanded set of examples. We currently use a language model that accepts any sequence of substrings of the examples, optionally separated by filler words, as well as sequences of digits. (For a small number of examples, a statistical N-gram model is ineffective because of low N-gram counts.) A detailed account of the recognition language model techniques used in the system is beyond the scope of this paper.

In the current implementation, actions are sequences of statements in the Java language. Constructors can be called to create new objects (e.g. a mail session object) which can be assigned to variables and referenced in other actions. The context interpreter loads the required classes and evaluates methods dynamically as needed. It is thus possible for an application developer to build a spoken interface to their target API without introducing any new Java classes. The system could easily be adapted to use action strings from other interpreted languages.

A key property of the process described above is that the application developer needs to know only the back-end API and English (or some other natural language).

4 Variant compilation

Different expansion methods can be used in the second step to produce variants v of an example e. In the simplest case, v may be a paraphrase of e. Such paraphrase variants are used in the experiments in section 7, where domain-independent "carrier" phrases are used to create variants. For example, the phrase *I'd like to* (among others) is used as a possible alternative for the phrase *I want to*. The context compiler includes an English-to-English paraphrase generator, so the application developer is not involved in the expansion process, relieving her of the burden of handling this type of language variation. We are also experimenting with other forms of variation, including those arising from lexical-semantic relations, user-specific customization, and those variants uttered by users during field trials of a system.

When v is a paraphrase of e, the adapted action a' is the same string as a. In the more general case, the meaning of variant v is different from that of e, and the system attempts (not always correctly) to construct a' so that it reflects this difference in meaning. For example, including the variant *show the message from Bill Wilson* of an example *read the message from John*, involves modifying the action `mailAgent.getWithSender("jsmith@att.com")` to `mailAgent.getWithSender("wwilson@att.com")`.

We currently adopt a simple approach to the process of mapping language string variants to their corresponding target action string variants. The process requires the availability of a "token mapping" t between these two string domains, or data or heuristics from which such a mapping can be learned automatically. Examples of the token mapping are names to email addresses as illustrated in the example above, name to identifier pairs in a database system, "soundex" phonetic string spelling in directory applications, and a bilingual dictionary in a translation application. The process proceeds as follows:

1. Compute a set of lexical mappings between the variant v and example e. This is currently performed by aligning the two string in such a way as that the alignment minimizes the (weighted) edit distance between them (Wagner and Fischer, 1974).

2. The token mapping t is used to map substitution pairs identified by the alignment ($\langle read, show \rangle$ and $\langle$*John, Bill Wilson*$\rangle$ in the example above) to corresponding substitution pairs in the action string. In general this will result in a smaller set of substitution strings since not all word strings will be present in

the domain of t. (In the example, this results in the single pair $\langle$`jsmith@att.com`, `wwilson@att.com`$\rangle$.)

3. The action substitution pairs are applied to a to produce a'.

4. The resulting action a' is checked for (syntactic) well-formedness in the action string domain; the variant v is rejected if a' is ill-formed.

5 Input interpretation

When an example-action context is active during an interaction with a user, two components (in addition to the speech recognition language model) are compiled from the context in order to map the user inputs into the appropriate (possibly adapted) action:

Classifier A classifier is built with training pairs $\langle v, a \rangle$ where v is a variant of an example e for which the example action pair $\langle e, a \rangle$ is a member of the unexpanded pairs in the context. Note that the classifier is not trained on pairs with adapted examples a' since the set of adapted actions may be too large for accurate classification (with standard classification techniques). The classifiers typically use text features such as N-grams appearing in the training data. In our experiments, we have used different classifiers, including BoosTexter (Schapire and Singer, 2000), and a classifier based on Phi-correlation statistics for the text features (see Alshawi and Douglas (2000) for our earlier application of Phi statistics in learning machine translation models from examples). Other classifiers such as decision trees (Quinlan, 1993) or support vector machines (Vapnik, 1995) could be used instead.

Matcher The matcher can compute a distortion mapping and associated distance between the output s of the speech recognizer and a variant v. Various matchers can be used such as those suggested in example-based approaches to machine translation (Sumita and Iida, 1995). So far we have used a weighted string edit distance matcher and experimented with different substitution weights including ones based on measures of statistical similarity between words such as the one described by Pereira et al. (1993). The output of the matcher is a real number (the distance) and a distortion mapping represented as a sequence of edit operations (Wagner and Fischer, 1974).

Using these two components, the method for mapping the user's utterance to an executable action is as follows:

1. The language model derived from context c is activated in the speech recognizer.

2. The speech recognizer produces a string s from the user's utterance.

3. The classifier for c is applied to s to produce an unadapted action a.

4. The matcher is applied pairwise to compare s with each variant v_a derived from a triple $\langle e, a, c' \rangle$ in the unexpanded version of c.

5. The triple $\langle v, a', c' \rangle$ for which v produces the smallest distance is selected and passed along with e to the dialog controller.

The relationship between the input s, variant v, example e, and actions a and a' is depicted in Figure 1. In the figure, f is the mapping between examples and actions in the unexpanded context; r is the relation between examples and variants; and g is the search mapping implemented by the classifier-matcher. The role of e' is related to confirmations as explained in the following section.

6 Confirmation and dialog control

Dialog control is straightforward as the reader might expect, except for two aspects described in this section: (i) evaluation of next-context expressions, and (ii) generation of

p (prompt): *say a mailreader command*
s (words spoken): *now show me messages from Bill*
v (variant): *show the message from Bill Wilson*
e (example): *read the message from John*
a (associated action): `mailAgent.getWithSender("jsmith@att.com")`
a' (adapted action): `mailAgent.getWithSender("wwilson@att.com")`
e' (adapted example): *read the message from Bill Wilson*

Figure 2: Example

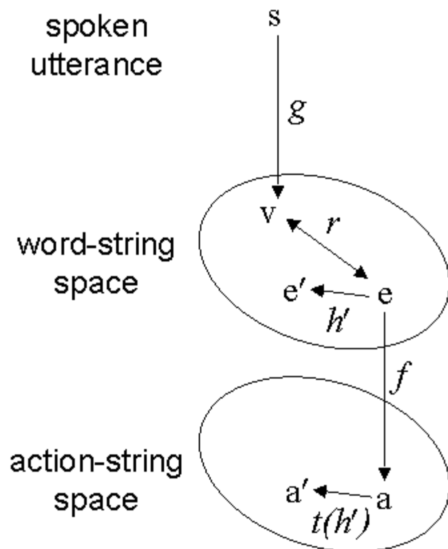

Figure 1: Variant Transduction mappings

confirmation requests based on the examples in the context and the user's input.

As noted in section 3 the third element c of each triple $\langle e, a, c\rangle$ in a context is an expression that evaluates to the name of the next context (dialog state) that the system will transition to if the triple is selected. For simple applications, c can simply always be an identifier for a context, i.e. the dialog state transition network is specified explicitly in advance in the triples by the application developer.

For more complex applications, next context expressions c may be calls that evaluate to context identifiers. In our implementation, these calls can be Java methods executed on objects known to the action interpreter. They may thus be calls on the back-end application system, which is appropriate for cases when the back-end has state information relevant to what should happen next (e.g. if it is an "intelligent agent"). It might also be a call to component that implements a dialog strategy learning method (e.g. Levin and Pieraccini (1997)), though we have not yet tried such methods in conjunction with the present system.

A confirmation request of the form *do you mean* e' is constructed for each variant-action pair (v, a') of an example-action pair (e, a). The string e' is constructed by first computing a submapping h' of the mapping h representing the distortion between e and v. h' is derived from h by removing those edit operations which were not involved in mapping the action a to the adapted action a'. (The matcher is used to compute h except when the process of deriving (v, a') from (e, a) already includes an explicit representation of h and $t(h)$.)

The restricted mapping h' is used instead of h to construct e' in order to avoid misleading the user about the extent to which the application action is being adapted. Thus if h includes the substitution $w \to w'$ but $t(w)$ is not a substring of a then this edit operation is not included in h'. This way, e' includes w unchanged, so that the confirmation asked of the user does not carry the implication that the change $w \to w'$ is taken into account in the action a' to be executed by the system. For instance, in the example in Figure 2, the word "now" in the user's input does not correspond to any part of the adapted action, and is not included in the confirmation string. In practice, the confirmation string e' is computed at the same time that the variant-action pair

(v, a') is derived from the original example pair (e, a).

The dialog flow of control proceeds as follows:

1. The active context c is set to a distinguished initial context c_0 indicated by the application developer.

2. A prompt associated with the current active context c is played to the user using a speech synthesiser or by playing an audio file. For this purpose the application developer provides a text string (or audio file) for each context in the application.

3. The user's utterance is interpreted as explained in the previous section to produce the triple $\langle v, a', c' \rangle$.

4. A match distance d is computed as the sum of the distance computed for the matcher between s and v and the distance computed by the matcher between v and e (where e is the example from which v was derived).

5. If d is smaller than a preset threshold, it is assumed that no confirmation is necessary and the next three steps are skipped.

6. The system asks the user *do you mean:* e'. If the user responds positively then proceed to the next step, otherwise return to step 2.

7. The action a' is executed, and any string output it produces is read to the user with the speech synthesizer.

8. The active context is set to the result of evaluating the expression c'.

9. Return to step 2.

Figure 2 gives an example showing the strings involved in a dialog turn. Handling the user's verbal response to the confirmation is done with a built-in yes-no context.

The generation of confirmation requests requires no work by the application developer. Our approach thus provides an even more extreme version of automatic confirmation generation than that used by Chu-Carroll and Carpenter (1999) where only a small effort is required by the developer. In both cases, the benefits of carefully crafted confirmation requests are being traded for rapid application development.

7 Experiments

An important question relating to our method is the effect of the number of examples on system interpretation accuracy. To measure this effect, we chose the operator services call routing task described by Gorin et al. (1997). We chose this task because a reasonably large data set was available in the form of actual recordings of thousands of real customers calling AT&T's operators, together with transcriptions and manual labeling of the desired call destination. More specifically, we measure the call routing accuracy for unconstrained caller responses to the initial context prompt *AT&T. How may I help you?*. Another advantage of this task was that benchmark call routing accuracy figures were available for systems built with the full data set (Gorin et al., 1997; Schapire and Singer, 2000). We have not yet measured interpretation accuracy for the structurally more complex e-mail access application.

In this experiment, the responses to *How may I help you?* are "routed" to fifteen destinations, where routing means handing off the call to another system or human operator, or moving to another example-action context that will interact further with the user to elicit further information so that a subtask (such as making a collect call) can be completed. Thus the actions in the initial context are simply the destinations, i.e. $a = a'$, and the matcher is only used to compute e'.

The fifteen destinations include a destination "other" which is treated specially in that it is also taken to be the destination when the system rejects the user's input, for example because the confidence in the output of the speech recognizer is too low. Following previous work on this task, cited above, we present the results for each experimental condition as

an ROC curve plotting the routing accuracy (on non-rejected utterances) as a function of the false rejection rate (the percentage of the samples incorrectly rejected); a classification by the system of "other" is considered equivalent to rejection.

The dataset consists of 8,844 utterances of which 1000 were held out for testing. We refer to the remaining 7,884 utterances as the "full training dataset".

In the experiments, we vary two conditions:

Input uncertainty The input string to the interpretation component is either a human transcription of the spoken utterance or the output of a speech recognizer. The acoustic model used for automatic speech recognition was a general telephone speech HHM model in all cases. (For the full dataset, better results can be achieved by an application-specific acoustic model, as presented by Gorin et al. (1997) and confirmed by our results below.)

Size of example set We select progressively larger subsets of examples from the full training set, as well as showing results for the full training set itself. We wish to approximate the situation where an application developer uses typical examples for the initial context without knowing the distribution of call types. We therefore select k utterances for each destination, with k set to 3, 5, and 10, respectively. This selection is random, except for the provision that utterances appearing more than once are preferred, to approximate the notion of a typical utterance. The selected examples are expanded by the addition of variants, as described earlier. For each value of k, the results shown are for the median of three runs.

Figure 3 shows the routing accuracy ROC curves for transcribed input for $k = 3, 5, 10$ and for the full training dataset. These results for transcribed input were obtained with BoosTexter (Schapire and Singer, 2000) as the classifier module in our system because we have observed that BoosTexter generally outperforms our Phi classifier (mentioned earlier) for text input.

Figure 4 shows the corresponding four ROC curves for recognition output, and an additional fifth graph (the top one) showing the improvement that is obtained with a domain specific acoustic model coupled with a trigram language model. These results for recognition output were obtained with the Phi classifier module rather than BoosTexter; the Phi classifier performance is generally the same as, or slightly better than, BoosTexter when applied to recognition output. The language models used in the experiments for Figure 4 are derived from the example sets for $k = 3, 5, 10$ (lower three graphs) and for the full training set (upper two graphs), respectively. As described earlier, the language model for restricted numbers of examples is an unweighted one that recognizes sequences of substrings of the examples. For the full training set, statistical N-gram language models are used (N=3 for the top graph and N=2 for the second to top) since there is sufficient data in the full training set for such language models to be effective.

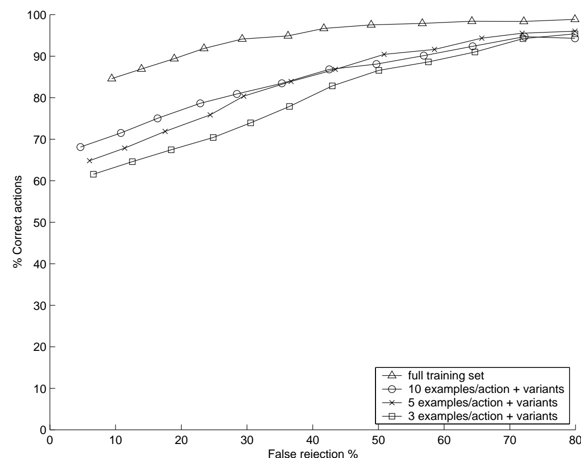

Figure 3: Routing accuracy for transcribed utterances

Comparing the two figures, it can be seen that the performance shortfall from using small numbers of examples compared to the full training set is greater when speech recog-

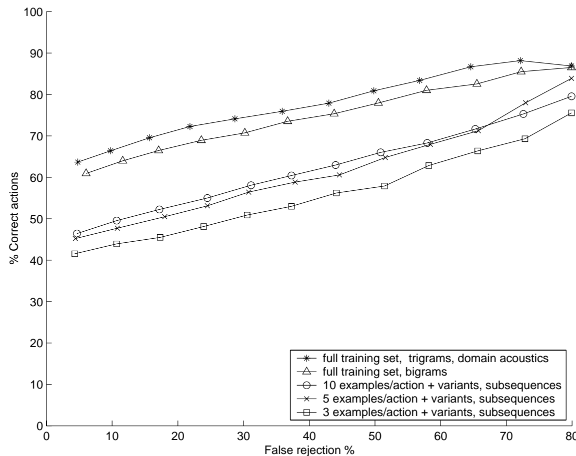

Figure 4: Routing accuracy for speech recognition output

nition errors are included. This suggests that it might be advantageous to use the examples to adapt a general statistical language model. There also seem to be diminishing returns as k is increased from 3 to 5 to 10. A likely explanation is that expansion of examples by variants is progressively less effective as the size of the unexpanded set is increased. This is to be expected since additional real examples presumably are more faithful to the task than artificially generated variants.

8 Concluding remarks

We have described an approach to constructing interactive spoken interfaces. The approach is aimed at shifting the burden of handling linguistic variation for new applications from the application developer (or data collection lab) to the underlying spoken language understanding technology itself. Applications are specified in terms of a relatively small number of examples, while the mapping between the inputs that users speak, variants of the examples, and application actions, are handled by the system. In this approach, we avoid the use of intermediate semantic representations, making it possible to develop general approaches to linguistic variation and dialog responses in terms of word-string to word-string transformations. Confirmation requests used in the dialog are computed automatically from variants in a way intended to minimize misleading the user about the application actions to be executed by the system.

The quantitative results we have presented indicate that a surprisingly small number of training examples can provide useful performance in a call routing application. These results suggest that, even at its current early stage of development, the variant transduction approach is a viable option for constructing spoken language applications rapidly without specialized expertise. This may be appropriate, for example, for bootstrapping data collection, as well as for situations (e.g. small businesses) for which development of a full-blown system would be too costly. When a full dataset is available, the method can provide similar performance to current techniques while reducing the level of skill necessary to build new applications.

References

H. Alshawi and S. Douglas. 2000. Learning dependency transduction models from unannotated examples. *Philosophical Transactions of the Royal Society (Series A: Mathematical, Physical and Engineering Sciences)*, 358:1357–1372, April.

H. Aust, M. Oerder, F. Seide, and V. Steinbiss. 1995. The Philips automatic train timetable information system. *Speech Communication*, 17:249–262.

Jennifer Chu-Carroll and Bob Carpenter. 1999. Vector-based natural language call routing. *Computational Linguistic*, 25(3):361–388.

J. Dowding, J. M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, and D. Moran. 1994. Gemini: A Natural Language System For Spoken-Language Understanding. In *Proc. ARPA Human Language Technology Workshop '93*, pages 43–48, Princeton, NJ.

A.L. Gorin, G. Riccardi, and J.H. Wright. 1997. How may I help you? *Speech Communication*, 23(1-2):113–127.

E. Levin and R. Pieraccini. 1997. A stochastic model of computer-human interaction for learning dialogue strategies. In *Proceedings of EUROSPEECH97*, pages 1883–1886, Rhodes, Greece.

Scott Miller, Michael Crystal, Heidi Fox, Lance Ramshaw, Richard Schwartz, Rebecca Stone,

Ralph Weischedel, and the Annotation Group. 1998. Algorithms that learn to extract information – BBN: description of the SIFT system as used for MUC-7. In *Proceedings of the Seventh Message Understanding Conference (MUC-7)*, Fairfax, VA. Morgan Kaufmann.

F. Pereira, N. Tishby, and L. Lee. 1993. Distributional clustering of english words. In *Proceedings of the 31st meeting of the Association for Computational Linguistics*, pages 183–190.

J.R. Quinlan. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA.

Robert E. Schapire and Yoram Singer. 2000. BoosTexter: A Boosting-based System for Text Categorization. *Machine Learning*, 39(2/3):135–168.

Eiichiro Sumita and Hitoshi Iida. 1995. Heterogeneous computing for example-based translation of spoken language. In *Proceedings of the 6th International Conference on Theoretical and Methodological Issues in Machine Translation*, pages 273–286, Leuven, Belgium.

V.N. Vapnik. 1995. *The Nature of Statistical Learning Theory.* Springer, New York.

Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, January.