Producing Dialog at MERL: problems in generation engineering

David D. McDonald
Mitsubishi Electric Research Laboratories
201 Broadway, Cambridge, Massachusetts
mcdonald@merl.com

Abstract

It is not necessary to have a full-scale NLG system in a dialog system that uses fixed-text templates for its responses to significantly improve in their cohesion and fluency. Given a flexible system, fixed phrases can be transparently replaced with pronouns or definite references. We describe how our DiamondTalk architecture lets us interpose our subsequent reference facility, Potemkin, between the synthesizer component and the Collagen collaborative dialog system where the texts originate.

1. Introduction[1]

MERL has a history of research on collaborative dialog. Our Collagen system (Rich & Sidner 1998), initially developed more than seven years ago, is based on long-standing work on the theory of discourse structure and collaboration by Candy Sidner, Barbara Grosz and their students (Grosz & Sidner 1990, Lochbaum 1998). Collagen has been the basis for more than a dozen demonstration systems, with more currently under development.

Making Collagen part of a spoken dialog system, however, has been a comparatively recent development. Adding speech recognition and synthesis into the mix, along with the goal of being able to evaluate an array of different recognizers and text-to-speech (TTS) systems, has led to a reevaluation of how Collagen should interact with its "I/O" components and to the development of a uniform architecture for spoken dialog, DiamondTalk, that has allowed us to freely 'plug-and-play' alternative recognizers, domain-specific semantic interpreters, special-purpose generators, and several synthesizers.

The flexibility of this architecture has made it possible to provide a simple fix to an particularly egregious problem inherent in the fixed-string generation facility we are using, namely that it has no ability to vary its descriptions in response to changes in the context. In cannot produce pronouns, definite noun phrases, or other reduced forms.

[1] This work would not have been possible without the collaboration of the other people in the speech and dialog group here at MERL: Bret Harsham, Neal Lesh, Chuck Rich Bhiksha Raj, Candy Sidner, Bent Schmidt-Nielsen, and Peter Wolf. Any errors or misrepresentations are my own.

We will start by describing the present state of our generation facilities, using examples from our ongoing spoken-dialog projects. From there we will briefly describe the DiamondTalk architecture. That will provide the setting for describing Potemkin. our 'last minute' system for repairing problems in cohesion.

2. Generation by composing fixed texts

There are many situations where the best choice of generation system is a suite of fixed texts. (We will use the term "fixed" texts to avoid the negative connotations of the word "canned" and to reflect the fact that our output is typically the result of composing pre-written text strings rather than been entirely written out.)

Consider the voice-driven navigation and telematics systems now appearing in high-end automobiles such as the Jaguar or the Infinity. The driver gives instructions to the car ("*radio tune 89.7*", "*I want to go to Narita station*"); the car responds by carrying out the action, asking for confirmation or clarification ("*please repeat*"), or giving instructions of its own ("*turn left in a quarter of a mile*"). The car has a relatively small set of things it needs to say (a recent evaluation called for a repertoire of about 300 utterances) and given the poor acoustic environment inside the car the best solution is to hire voice talent to pre-record everything that the car may need to say.

2.1 Letting in some grammar

The simplicity of a completely fixed output corpus breaks down as soon as the task requirements include the need to incorporate references to an open-ended set of entities (e.g. street addresses). Often just replacing a variable in the text with the value it has in this instance will suffice. However, if this "direct replacement" code is being written by a research team, the temptation to add just a 'little bit' of grammar becomes too great to resist.

We can see this in systems dating back at least to Winograd's SHRDLU (1972). People whose concentration is on problems other than generation (in our case dialog, discourse structure, task modeling, agents, and tutoring) will nevertheless want to embellish their substitution systems with code that adds plurals, puts verbs in past tense, adds markup, etc. This becomes a slippery slope.

Having a facility that lets you increase the fluency of your output by writing more code makes it easy to postpone, perhaps indefinitely, the move to a 'real' generation system with an independent grammar, a planning phrase, and microplanning capabilities.

2.2 Collagen's Glosser

This is the situation here at MERL. Thanks to some very clever work by Egon Pasztor, we have a rich programming language (a suite of Java classes) for providing "glosses" for the terms in our task models. Taking an example from our speech-driven forms-filling system FormsTalk, suppose the user has said "*617 621 7585*". Based on the pattern of digits, the speech recognition system knows that it has heard a phone number, but the pragmatic component (Collagen) knows that it does not have enough information to determine which of the two empty phone number fields in the form the user intended it to fill. In this situation our agent uses the code below to construct the utterance "*Is 617 621 7585 your home phone or work phone?*".

```
public Gloss glosser (WhichField which) {
  Gloss g = new Gloss();
  translateVerb(g, "collagen.verb.is")
    .append(" \'")
    .append(render(which.getFiller()))
    .append("\' ");
  translateSp(
     g, "collagen.actor.possessive.other")
     .append(which.getFields());
  questionize(g);
  prependSubjectAndVerb(g, which);
  return g;
}
```

(This is a Java method that takes an ambiguous field of the form as its argument. Terms like collagen.actor. possessive.other pick out properties in a resource file, in this case the string "*your*". The translate methods looks up their values: translateVerb takes tense into consideration and translateSp saves a common line of code by adding a space.)

Our 'gloss-based' generator, while unlikely to ever make the cut for a generation conference, has much to recommend it. Once you become fluent in its programming idioms, it lets you go well beyond what can be achieved by direct string replacement. The gloss data structure records where objects were referenced within the string (e.g. the objects representing the phone number and the two fields), and glosses can be presented in a viewer where you can see what these objects were and edit them. Using symbols as the intermediaries to strings (collagen.verb.is) makes glosses easily localized to a particular language by changing the interpretation of the symbols. At a click of a button the same utterance can presently appear as English, French, or Japanese. It can also be rendered in the underlying planning language or even its raw hexadecimal, which is invaluable for debugging.

Never the less, our glossing facility has two severe limitations. The first is the sort that would be cured through the use of a grammar with good coverage. Suppose you were not satisfied with the phrasing of the system's response and wanted to have it say "*Is 617 621 7585 your home phone or your work phone?*", repeating the possessive on either side of the conjunction. In principle this could be done in gloss code, but in practice it would be too difficult to be worth the effort. On the other hand, if the production of the text were mediated by a grammar, it might require just adding a single feature.

The other problem is more noticeable and more severe. Glosses are context free—there is no means for the discourse context to influence what is said. The gloss code will always replace objects with the same strings regardless of whether this was an object's first reference or tenth, whether it is in focus or deducible from context. While replacing the glossing facility with something more linguistically principled is too extensive a project to do in the short term, the flexibility afforded by the DiamondTalk Architecture permits us to immediately go after a particular class of 'low-hanging fruit': the production of subsequent references to concrete objects, and to do so in a way that is transparent to other system operations.

3. The DiamondTalk Architecture

Our choice of what to focus on in our work has mandated that rather than build our own recognizers or synthesizers from scratch we adapt systems that are 'taken off the shelf. This leads to a focus on evaluating alternatives (at this point we have been looking at more than five recognizers and six text-to-speech systems) and on using machinery that lets us readily swap one for the other within the same application.

To this end, DiamondTalk is defined by a set of interfaces that its components must satisfy, and by a message-passing protocol that links their activities. Neither the choice of implementing component nor the links between components are fixed: they are set at launch time by wiring routines that connect event sources to event listeners according to the specification given in the launch command.

The toplevel of DiamondTalk consists of four components as show in Figure One. The arrows show only the major aspects of the information flow. Dotted arrows indicate the transfer of state information.

The structure and message protocols within subcomponents can be whatever the particulars of the implementing systems demand. The contract on the SpokenInput component, for example, is only that it pass on to the Manager a domain-specific semantic interpretation of what it heard. It is immaterial to the operations downstream whether the speech source was a person speaking live into a microphone, a stored WAV file, or a file of text strings.

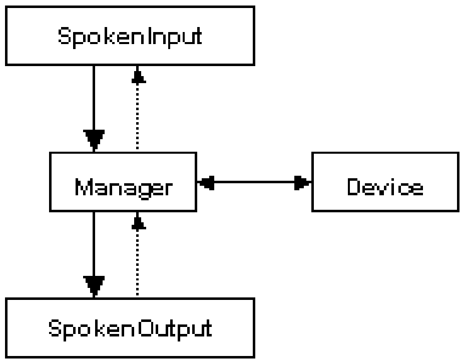

Figure One: DiamondTalk

3.1 Spoken Output

Similarly the internal composition and operation of the SpokenOutput component is completely independent of other components' assumptions. The system's input and output languages don't even have to be the same. The input could be English and the output Japanese. Indeed an option we are considering for debugging our tutor is for the output to be in two languages at once: spoken Japanese and written Japanese and English.

Our SpokenOutput component breaks down into a Generator component and a Synthesizer component. The Manager issues a statement of what content is to be said which is listened for by the Generator. The Generator is responsible for determining the structure and wording of a text that can communicate this content in the present situation. In terms of the usual model of a NLG system, it is responsible for microplanning and surface realization. In fact what it does is apply the appropriate glossing methods (recursively) to the structured object that is passed to it from the Manager.

The generator issues a message containing that information that is listened for by the Synthesizer component which is responsible for uttering it. If the Synthesizer is a TTS engine this is just the sequence of words with some markup. The string is the output of the glossing methods.

Other components than just the Synthesizer can listen to the Generator's output. This is how, for example, we can have the text both spoken and printed to a trace window. Since the inter-component wiring is explicit, we can also arrange for the output of the Generator to be 'intercepted' by a component that satisfies the interfaces of both the Synthesizer and Generator, modifying the Generator's original output and then issuing the new text on to the Synthesizer. This is how the Potemkin system achieves the transparency alluded to earlier. It emulates a Synthesizer to get the output of the Generator, and it emulates a Generator to issue revised strings to any connected listeners, principally the Synthesizer.

The Synthesizer components we are using are not simply wrappers around text-to-speech systems, though that is a significant part. As part of our work on hosting (Sidner & Dzikovska 2002), we have a stationary, foot-high robot shaped like a penguin that we call Mel. Mel is equipped with face recognition and audio localization systems, and it has been instrumented to use the two degrees of freedom in its trunk and head to attend to the face or voice of people in the vicinity. Part of what Mel can do is use its head to nod at appropriate moments while it is speaking. To do this we use a copy of the BEAT system developed at the MIT Media Lab (Cassell et al. 2001). BEAT is driven by a simple model of the distribution of given and new information in the text, as indicated by markup tags, which we supply by simple observations of the status of objects within Collagen's focus spaces. We get a rough timing specification from the TTS of the words that will be spoken, augment the text-markup with that and the given/new information, and then feed this as an independent stream to Mel's control system. We have also set up Mel to use its beak as a pointer to show the user what to look at on a table display of a complex control panel, where it is working from the same data stream as we use with our mouse-based pointer on a display screen.

3.2 Collagen

The Manager component in DiamondTalk is a wrapper around the core components of the Collagen system, as shown in Figure Two. Collagen has been extensively described elsewhere (e.g. Rich et al. 2001), but in brief, the Discourse component makes use of a Task Model to compute the pragmatic interpretation of what the user has just said or done. This updates the Discourse History and leads the Agent to decide what it should say or do in response. The Discourse and History are then updated to reflect what the Agent does as it carries out its action.

While we refer to the component as a 'manager', implying that its role is to coordinate the efforts of other components of the system (compare the notion of a 'hub' in Polifroni & Seneff 2000), Collagen is better thought of as a full participant in the conversation, trying to collaborate with the human user in the completion of some task. (The wrapper around Collagen in DiamondTalk does do some strictly coordinating activities, such as orchestrating push-to-talk and barge-in, and translating between representations.)

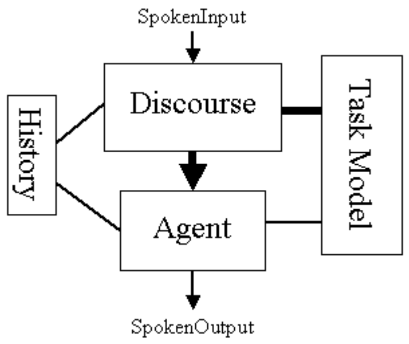

Figure Two: Collagen

4. Potemkin

The downside of using fixed texts is of course that however fluent a text might be in isolation, it will always come out the same way each time it is used, even when it shouldn't. We developed the Potemkin component to address one particular instance of this problem, generating subsequent references to objects. Here we will describe where Potemkin fits in, its assumptions, its choice of representations and decision criteria, and provide a short example.

Reference generation has developed a rich literature over the years (e.g., van Deemter 2002, Dale & Reiter 1995, Bateman & Teich 1995, Dale 1992). The focus of most of that work, however, is on how to construct contextually appropriate 'initial' references—what you say the first time an object is mentioned. Our concern, in contrast, is with how the object is realized the next time it is referenced ("subsequent reference"), particularly with cases where a pronoun would not be appropriate. This kind of work has received more attention in the information extraction community than in generation (but see, e.g., Callaway & Lester 2002 or McCoy & Strube 1999), yet it is an area of considerable importance in generation since well fashioned subsequent references are a key aspect of what makes a text fluent.

4.1 The problem

The problem with a generation system that has no subsequent reference capability can be seen in an example from our tutoring system, Paco (Rickel et al. 2002). Figure Three shows a short except from a session between Paco, **P**, and a student, **S**, who is being show how to find their way around the (simulated) control panel of a power plant. This is what we get without employing Potemkin.

P1: Let's navigate to the screen that contains the turbine speed indicator. Notice that the Alarm button is red, indicating that we are viewing the Alarm panel. Ok?

S1: Ok.

P2: *Press the Operation button.*

S2: *< user presses the button >*

P3: *Right. Now notice that the Operation button is red, indicating that we are viewing the operation panel. Ok?*

S3: Ok.

P4: Now press the Operation Menu button.

Figure Three: Excerpt from tutoring dialog

When you hear these texts spoken by a text-to-speech system, even one that is state of the art, you are immediately struck by the fact that it doesn't say "*the button*" in P3. This is a failure in cohesion (Halliday & Hasan 1976) that come from it only having one way to say "the Operation button" (P2, P3). Even though the gloss methods in Paco are compositional and often quite sophisticated (sentence P1, for example, was produced in four chunks, including the deliberate use of "*let's*" as a discourse marker), they only have one way to refer to each of the objects in the power plant control panel.

One could, perhaps, provide the needed alternatives on an ad-hoc basis, but with the glossing machinery already stretched to its limit as a programming system, its has no practical way to access the rich contextual representations of the discourse that are in other respects central to Collagen's design and operation. Consequently alternatives can only be used randomly. (Paco does do this for acknowledgements and channel checkers.) This led us to start a small project (well under a man-week as this is written) to develop the Potemkin system.

4.2 Changing the wiring

The loose-coupled nature of the components in Diamond-Talk lets us interpose a new component between the Generator and Synthesizer without having to touch the code in either of them, only the code that wires them together. The Potemkin component satisfies the interfaces of both a Generator and Synthesizer, letting us wire the output of the regular Generator to the input of Potemkin, viewed via its synthesizer interface, and then wire the output of Potemkin, viewed via its generator interface, to the standard input of the Synthesizer.

Potemkin's design relies on two facts about the Paco tutor:

(a) that it works in a closed-world where the identity of every object that might be talked about is known in advance, and
(b) that the fixed-text generator (glossing) will produce the very same text string every time a particular parameter value is mentioned.

This permits Potemkin to have instances of strings, such as "the Operator button" serve as proxies for what might in other circumstances be richly typed data structures maintained explicitly by the control room simulator. (The simulator does have structures for this and other objects, but with only enough type information to support its displays (e.g. a type for 'button' but not for 'Operator', which is just a string). In addition, typing and object identities are modified (wrapped) when these objects are projected into Collagen for reasoning within the task model.)

Essentially what Potemkin does is establish a parallel system of representation—one that is intended to capture just the information that is relevant to subsequent reference generation. This lets us experiment with what needs to be represented and with what information should be taken into account for this task, all without having to modify other regions of the code every time one wants to try out some alternative.

4.3 Expressive types

The representation used in Potemkin employs two systems of types. An object's 'intrinsic type' reflects its identity as a instance of a class or a natural kind. These types are the sorts of things that would be deep sources for common nouns: "*button*", "*panel*". Their role in Potemkin is to let us keep tract of 'distracters', e.g. what other buttons have been mentioned recently that might be confused with the one we are about to mention.

Every object also has an 'expressive type', that governs how it can be realized. These are a minimal treatment of the concept developed by Meteer (1992) to capture the fact that an object's possible realizations fall into systematic classes that are generic to a language and which facilitate reasoning about what choices to make when generating. In Meteer's theory, expressive types are used during microplanning to reason abstractly about the options for composition. For example they facilitate the handling of lexical gaps: Given an object with an expressive class that includes both 'make NP' and 'VP' alternatives ("make a decision", "decide") and the requirement to compose it with some version of the modifier 'important', only one of the two alternatives will support it since the class for 'important' has no realization as an adverb.

As we are using them in Potemkin, the expressive types only incorporate information about options for subsequent reference, where the goal is to produce the most reduced realization that is licensed by the context. To this end, the types are organized into a lattice as shown in Figure Three. This is a subsumption lattice organized from general to specific. Any object that can be realized by an expressive type low in the lattice could also, conditions permitting, be realized by any of the parents of that type. Higher types in the lattice produce more reduced, more cohesive, realizations than their daughter types. The 'individuated' expressive type produces pronouns.

Note that this set of expressive types is by no means complete (it is probably low by several orders of magnitude), but it has proved sufficient so far for our initial dialogs in the power plant domain.

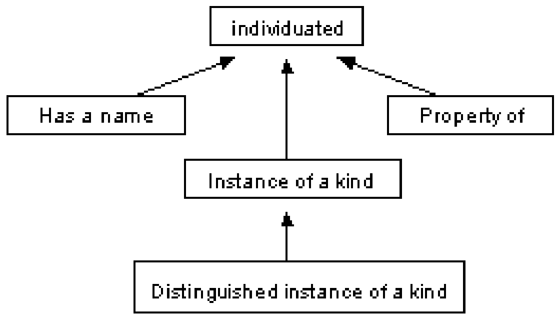

Figure Four: Expressive Types

Returning to our example, we have classified 'the Operation button' as having the expressive type 'distinguished instance of a kind' since the term 'Operation' distinguishes it from other instances of the kind 'button'. 'Operation' is represented as its own intrinsic type. To set up the universe of objects that we need in order to support subsequent reference realizations of 'the Operation button' in our example (the Potemkin village as it were), we execute the three statements below, which create the objects we need.

```
defineKind("button");
defineKind("Operation");
create("the Operation button",
       kindNamed("button"),
       new EType.DistinguishedIOAK(
            kindNamed("button"),
            kindNamed("Operation")));
```

The defineKind method creates a new intrinsic type with the indicated name; all intrinsic types get the expressive type 'has a name'. The create method makes an object of the indicated kind whose full initial reference text is given as the first argument. Note that this is the identical string by which it will be given by the glossing machinery. The method also makes an instance of the object's expressive type, 'distinguished instance of a kind', where we specify what the kind and the distinguisher are.

4.4 How it works

Potemkin's procedure is straightforward. As the Agent makes its decisions about what to say, a succession of fixed texts, the results of glossing facility, will be issued from the Generator component to any listening components (Potemkin and a monitoring window). Potemkin examines each string that it receives for substrings that match any of the strings of the objects it knows about. When there is a match, it uses the matched string to looks up the corresponding object, calls the object's realize method to receive the string it should use in this instance, makes the substitution, and then continues its search. When the whole text string has been examined it is issued to the TTS engine in the Synthesis component just as though the Generator component had done it. This means that the subsequent reference machinery in Potemkin is run incrementally, left-to-right, in the natural order of the text and can consequently assume that all of the relevant backward-looking context has been established when it sets about to make its decisions object by object.

Given the expressive type machinery, with the alternatives organized into a subsumption lattice, the algorithm for producing subsequent references is very simple. First we check whether the object has appeared in the discourse history. If it has not, then this is an initial reference and we use the fixed text as came out of the gloss. Otherwise, we walk up the object's line in the lattice starting with the expressive type that the object was assigned when it was defined. Each type has a method that examines the object and the discourse context to determine whether is licensed for use. We scan for the most general type that is licensed in this particular instance and take the realization that it produces.

This design has considerable economy. The decision criteria are given just once, regardless of how many objects may need to use them. Its production machinery is compositional wherever possible, with lower, more specific types assembling their larger realizations out of the smaller parts provided by the higher, more generally applicable types. To use this technique, each object (or object type) just has to be assigned an expressive type by creating an instance of the type in order to indicate, e.g., what the object is a kind of or what other objects acts as its distinguisher from other objects of that kind. The result is quite lightweight.

The licensing checks for the expressive types—their decision criteria for whether they can provide the realization—are only preliminary as this is being written and they are very simple. They suffice for the few tutoring scenarios we have looked at so far, but whether they will continue to produce nice results in other tutoring situations is hard to determine. We do know that since glosses produce only strings of fixed text and not syntactic structures that the criteria are likely to be quite brittle. A case in point is the licensing criteria for Individuated, the source of pronouns. It is only licensed if the previous instance of the object was as the very first thing in the sentence. That is a weak substitute for knowing whether it was the grammatical subject, which in turn is a weak substitute for knowing that the object is intended to be in focus, something we will be able to represent when we have adapted Collagen to directly support generation; see below.

To look briefly at Potemkin in action, we repeat here the portion of the example Paco dialog where the object 'the Operation button' occurred:

P2*: Press the Operation button.*
S2*: < user presses the button >*
P3*: Right. Now notice that the button is red, indicating that we are viewing the operation panel. Ok?*

The instance in P2 is the first occurrence of OB in the dialog, so the full text that the gloss produces is used ("*the Operation button*"). In its second instance, P3, Potemkin's operations take over. It was in object position in P3, so the top option in the expressive type lattice is not licensed and we ask whether the next type down, 'Instance of a kind', is licensed. Its criteria is recency. It is licensed so long as the last instance of the object was part of the last Agent/User exchange (P2, S2). In this example it was part of the Agent's last utterance (P2) so the type is licensed. This being the case, we employ the realization method that comes with 'instance of a kind'. Since we are only dealing with strings, this is just a matter of appending the name of the kind ("*button*") after the word "*the*". Crude but effective.

5. Future directions

As we have said, Potemkin is only intended as an easily developed, stop-gap measure that addresses one particular nagging problem in the texts that our glossing facility produces. Our eventual goal is to develop a generator that is specifically tuned to the requirements of spoken dialog and the capabilities of the Collagen system. The question is how to get there via an evolutionary path that is minimally disruptive. We cannot, for example, simply choose to expunge the glossing facility from our code, if for no other reason that it would instantly break the fourteen Collagen demos that we maintain. And indeed there are situations where we would not want to.

5.2 Predicate-argument structure

We need more flexibility in composing minimal (fixed) phrases: the composition needs to reflect their information status in the discourse and the intentions of the speaker (i.e. the Agent component within Collagen). We also need to annotate the texts with their linguistic structure so that we can guide the prosodic judgments of a speech synthesizer to reflect the information structure.

To do this, we are considering having the authors of (future and ongoing) glossing methods include a specification of the text's predicate-argument structure along with their fixed glosses. The predicates would subsume the constant portions of the texts, and the arguments, as before, would be references to objects inside Collagen's task model. The predicates would then provide a substrate for the linguistic structure.

The intention is just to abstract away from the literal wording that would otherwise have been used—to deploy symbols in lieu of text-strings. Unless pushed to it by additional goals, the predicates do not need to be drawn from a general ontology or an inter-lingua (e.g. along the lines of the SENSUS project (ISI 2002)). Our earlier glossing example ("*Is 617 621 7585 your home phone or your work phone?*") could be assigned 'simple-copular-question'. (There is not point in abstracting out relations like ‘question’ without a grammar in place to mediate between the predicates and their surface forms.)

One such goal however might be to provide the information needed to drive a real microplanner. (This is the term in NLG for a component that manages tactical issues such as how minimal phrases are combined, how sentence lengths and styles are determined, and choices among semantically comparable words.) We have been talking to Charles Callaway about possible collaborations where we would incorporate his microplanning techniques (Callaway & Lester to appear).

5.3 Introducing surface syntax

Initially the predicates will just be syntactic sugar over the individual fixed texts. The move to add linguistic descriptions will take place slowly because there are research issues involved. We do have access to a large TAG grammar that is optimized for language generation and would have ample coverage for the kinds of constructions that our dialog systems tend to use, but it is not clear that this would be the correct choice since its design reflects the surface syntax of a text, not its information structure.

That grammar was used as the basis of quite reasonable speech (for that time) using a Klatt-based rule-based synthesizer by following Pierrehumbert’s original rules and fitting the vowel pitches to a catenary curve between the start and end points of the major syntactic phrases (Rubinoff 1986). However, since we have the opportunity to develop an interesting model of information-structure because of the resources in Collagen, we are looking at theories of how information structure is related to prosody such as Mark Steedman’s (e.g. Steedman 2002), and this could very well lead to different choices.

Part of the problem is that to produce the best results, the choice of linguistic annotation on the predicates should be tailored to the choice of speech synthesizer. Unfortunately this choice is not obvious. On the one hand, to get the best quality speech with minimal effort, particularly for audiences of non-native speakers of English, we should use one of the commercial TTS systems, such as the offerings from AT&T[2] or Rhetorical Systems. [3] Our experiments show that they do indeed sound strikingly like a person if the choice of prosodic tune is not an issue.

However the choice of tune is very much an issue in the concept-to-speech word we intend to do (for groups with comparable issues see Alter et al. 1997). Consider our examples from the tutoring domain. The agent’s first turn (Figure Three, P1) was this sentence: “*Notice that the Alarm button is red, indicating that we are viewing the Alarm panel.*”. ‘Alarm’ is given information the second time it is mentioned, and a person uttering that sentence is very likely to use a tune for “*the Alarm panel*” that reflects that information structure by putting the phrasal stress on “*panel*” and using a low or level pitch for “*Alarm*”. The off-the-shelf TTS systems, on the other hand, because they are using a default model for the structure of a simple nominal compound, put the stress on “*Alarm*”, which is completely at odds with the discourse facts and sounds jarring to the attentive listener. It would be natural to add this prosodically realized option to the realization choice of the ‘distinguished instance of a kind’ expression type, but that will only be possible if the synthesizer supports it.

If we are willing to live with artificial sounding output we can use an off the shelf synthesis-by-rule system that we have experimented with, the Eloquence TTS developed at OGI that is available from SpeechWorks[4] and elsewhere. It accepts a very rich markup language that would appear to allow one to express anything that can be represented using ToBI markup (Pierrehumbert & Hirschberg 1990). There is also of course Festival,[5] and the Java-based FreeTTS[6] that can efficiently drive Festival voices. These have been used to great advantage by groups that have been able to devote the resources to developing application-specific concatenative voices (e.g. Swartout et. al 2001).

In this paper we have described our architecture for spoken dialog systems and how it lets us transparently insert a new component, a specialist in the generation of subsequent references, between a generator that composes fixed-text utterances and the synthesis component. We have also laid out our future plans, which are to adapt the rich model of discourse in the Collagen collaborative dialog system to the needs of a generator that is targeted to the demands and capabilities of today’s speech synthesizers. We are especially fortunate because here at MERL we have a speaker that actually knows why it is saying what it does—it has real intentions towards its audience. This is new territory for most all work in natural language generation, and we hope to take advantage of it.

References

Alter, K, H. Pirker, and W. Finkler eds. (1997) Concept to Speech Generation Systems: Proc. of the ACL workshop, July 11, 1997, Madrid.

Bateman, J.A. & E. Teich (1995) Selective information presentation in an integrated publication system: an application of genre-driven text generation, Information Processing and Management: Special Issue on Summarizing Text, 31(5), 753-768, September.

[2] http://www.naturalvoices.att.com/

[3] http://www.rhetoricalsystems.com/

[4] http://www.speechworks.com/products/tts/eti.cfm

[5] http://www.cstr.ed.ac.uk/projects/festival/

[6] http://sourceforge.net/projects/freetts/

Cassell, J,. Vilhjalmsson, H., Bickmore, T. (2001) BEAT: the Behavior Expression Animation Toolkit, Proc. SIGGRAPH 2001, New York, 477-486.

Calloway, C.B. & J.C. Lester (2002) Pronominalization in Generated Discourse and Dialog, Proc. 40th Annual Meeting of the ACL, University of Pennsylvania, 88-96

Calloway, C.B. & J.C. Lester (to appear) Narrative Prose Generation, AI Journal.

Dale, R. (1992) Generating Referring Expressions, MIT Press.

Evans, R., P. Piwek, and L. Cahill (2002) What is NLG?, Proc. 2nd International Natural Language Generation Conference, Arden Conf. Center, Harriman New York, July 1-3 2002, 144-151

Lochbaum, K.E. (1998) A collaborative planning model of intentional structure, Computational Linguistics 24(4) 525-572.

Grosz, B. & C. Sidner (1990) Plans for discourse, in P.R. Cohen, J. Morgan, & M.E. Pollack (eds.) Intentions in communication, chapter 20, 417-444, MIT Press.

Halliday, M.A.K. & R. Hasan (1976) Cohesion in English, Longman.

ISI (2002) http://www.isi.edu/natural-language/resources/sensus.html

McCoy, K.F. & M. Strube (1999) Taking time to structure discourse: Pronoun generation beyond accessibility, Proc. 21st Conf. Cognitive Science Society, Vancouver, 378-383.

Meteer, M. (1992) Expressibility and the Problem of Efficient Text Planning, Pinter.

Pierrehumbert, J. & J. Hirschberg (1990) The Meaning of Intonational Contours in the Interpretation of Discourse, in Cohen et al. (eds.) Intentions in Communication, MIT Press 271-323.

Polifroni, J. & S. Seneff (2000) Galaxy-II as an Architecture for Spoken Dialog Evaluation, Proc. Second International Conference on Language Resources (LREC), Athens, Greece, May 31-June 2, 2000.

'Ongoing Ontology projects' http://www.cs.utexas.edu/users/mfkb/related.html

Reiter, E., R. Robertson, and L. Osman (1999) Types of knowledge required to personalize smoking cessation letters, in Proc. AIMDM'99, 389-399.

Rich, Ch., C. Sidner & N. Lesh (2001) COLLAGEN: Applying Collaborative Discourse Theory to Human-Computer Interaction, AI Magazine 22(4) Winter, 15-25.

Rickel, J., N. Lesh, C. Rich, C.L. Sidner, and A. Gertner (2002) Collaborative Discourse Theory as a Foundation for Tutorial Dialog, International Conference on Intelligent Tutoring Systems (ITS), Vol. 2363, 542-551, June (Lecture Note in Computer Science).

Rubinoff, R. (1986) Adapting mumble: Experience with natural language generation, Proc. AAAI-86, 1063-1068.

Sidner, C. & M. Dzikovska (2002) Hosting Activities: Experience with and Future Directions for a Robot Agent Host, Proc. IUI'02, January 13-16, San Francisco.

Steedman, M. (2002) Information-structural Semantics for English Intonation, draft 2.1 on www.information.cs.ed.ac/~steedman/papers.html

Swartout, W, R. Hill, J. Gratch, W.L. Johnson, C. Kyriakakia, C. LaBore, R. Lindheim, S. Marsella, D. Miraglia, B. Moore, J. Morie, J. Rickel, M. Thiebaux, L. Tuch, R. Whitney & J. Douglas (2001) Toward the Holodeck: Integrating Graphics, Sound, Character, and Story, Proc. 5th Conf. on Autonomous Agents, Montreal, June 2001.

Welty, C. and B. Smith (eds.) (2001) Formal Ontology in Information Systems, ACM Press.

Winograd, T. (1972) Understanding Natural Language, Academic Press.