Natural Language Engineering **1** (1): 000–000. Printed in the United Kingdom
© 1998 Cambridge University Press

An Architecture for a Generic Dialogue Shell

J. ALLEN, D. BYRON, M. DZIKOVSKA,
G. FERGUSON, L. GALESCU, A. STENT
*Department of Computer Science, University of Rochester
Rochester, NY 14627*

(*Received 1 March 2000*)

1 Introduction

Different researchers use the term "dialogue system" in different ways with little in common beyond the belief that a dialogue system necessarily interacts with a human. For some, a dialogue system involves some mechanism to constrain the interaction, such as a script that specifies system prompts and user responses. The most common examples of this type of system in use are the telephone interfaces based on keyed or spoken menu selection (*e.g.*, "if you want your account balance, press or say 1," and so on). The goal here is to *restrict* the user's options in the interaction to simplify the language processing. For others, dialogue work aims to produce machines that can mimic human conversation. Work in this area aims to provide intuitive access to a wide range of applications, either over the telephone using voice only, or with multi-modal interaction at a computer workstation. The goal of this approach is to *expand* the user's options in the interaction.

We have the latter goal. We believe that such systems could be both feasible and cost-effective within the next decade. While speech-driven "menu-style" dialogue systems are growing in use today, they have their limitations. Unless the user needs to perform the most common and expected tasks, and can adapt to the preset method of performing those tasks, such interfaces can be a source of frustration. In addition, it appears that more complex tasks will be impossible to perform using such limited dialogue interaction. To support effective dialogue, the input language must be sufficiently general to express a wide range of different goals and courses of action. Effective interaction also requires an ability to support "mixed-initiative" interaction. The user must be able to "lead" the conversation in ways that best accomplish their goals, and the system should be able to take the initiative to speed up the solution when opportunities arise. In all but the simplest tasks, the system must be able to perform intention recognition (to determine what part of the task is currently being performed) and it be able to track the flow of topic as it changes throughout the dialogue. The system must also be able to plan responses that can only be realized incrementally over several turns in the dialogue. Finally, the system must be concerned with grounding: it must efficiently signal that the user was understood, and it must recognize when the user signals understanding or non-understanding.

We are, however, faced with a dilemma. Allowing unrestricted natural language dialogue would appear to require full human conversational competence, which does not seem feasible in the foreseeable future. We believe this argument is flawed, and that the required levels of conversational competence can be achieved in applications in the foreseeable future and at reasonable cost. The reason is that applications of human-computer interaction all involve dialogue focussed on accomplishing some specific task. We believe that the goal-seeking nature of such conversations naturally creates a specific genre of conversation, which we call *practical dialogues*. A practical dialogue could involve tasks such as performing a simple transaction (*e.g.*, ordering some merchandise), information-seeking (*e.g.*, determining the arrival of flights, accessing medical information), engaging in problem solving (*e.g.*, designing a kitchen), command and control (*e.g.*, managing the response to a natural disaster), or tutoring (*e.g.*, teaching basic concepts of mathematics). Our optimism depends on two hypotheses. The first concerns the complexity of practical dialogue:

> *The Practical Dialogue Hypothesis*: The conversational competence required for practical dialogues, while still complex, is significantly simpler to achieve than general human conversational competence.

Even though a practical dialogue system might be possible to construct, however, it might still be too *expensive* to construct in practice. Again, based on our experience of building experimental systems in a number of domains, we believe that it will not. This suggests our second hypothesis:

> *The Domain-independence Hypothesis*: Within the genre of practical dialogue, the bulk of the complexity in the language interpretation and dialogue management is independent of the task being performed.

If these hypotheses are true, then it should be possible to build a generic dialogue shell for practical dialogue. By "dialogue shell" we mean the full range of components required in a dialogue system, including speech recognition, language processing, dialogue management and response planning, built in such a way as to be readily adapted to new applications by specifying the domain and task models. This paper documents our progress and what we have learned so far based on building and adapting systems in a series of different problem solving domains. The first system, TRAINS, was built in 1995 (Allen *et al.* 1995; Ferguson *et al.* 1996), and the task was to find efficient routes for trains in the northeast United States (the so-called "TRAINS domain"). In 1996 and 1997, we evaluated the TRAINS system (Stent & Allen 1997). The users had to find efficient routes for a group of trains (typically three trains) and avoid problem areas as they were discovered (*e.g.*, congestion, tracks out). In a controlled experiment involving 80 sessions with 16 users, each of whom received less that three minutes of training, over 90% of the dialogue sessions resulted in successful plans without any intervention at all from the experimenter.

Based on this experience, we designed a new dialogue system architecture. One of the main goals of this effort was to separate functionality and enhance postability to new domains. The result was TRIPS, The Rochester Interactive Planning System

Table 1. *Summary of TRIPS task domains*

Domain	Date	Task	Goal	Status
TRAINS	1995-7	Finding efficient routes for trains	Robust performance on a very simple task	Robust performance (> 90% success rate)
PACIFICA	1997-8	Evacuating people from an island	Robust performance in a task requiring explicit planning	Demonstration system supports untrained users
CPoF	1998	Deployment of troops in a military situation	Scripted demonstration in a military relevant task	Scripted interaction only
Monroe	1999–	Coordinating responses to emergencies in Monroe County, NY	Robust performance on a dynamic, mixed-initiative task involving planning, monitoring and replanning; larger domain	In development for robustness evaluation
AMC	1999–	Planning airlifts using an airlift planning system	Demonstrate ability to use third-party planning systems; emphasis on agent technology	Initial demonstration completed, work on extensions continuing
Kitchen	Planned	Planning kitchen design	Robust performance on a significantly different task; multilingual experiments	Planned for development

(Ferguson & Allen 1998). TRIPS is designed to support plan-based tasks, and we have built versions of the system in several domains. TRIPS-PACIFICA involves evacuating people off an island in the face of an impending hurricane and is robust for naïve users. TRIPS-CPoF was a limited scripted-only demonstration system that involved planning the deployment of troops. TRIPS-AMC involves planning airlifts and investigates how "third-party" back-end systems could be integrated into the TRIPS architecture. The Monroe domain, under current development, involves coordinating emergency vehicles in response to simulated 911 calls and serves as our current experimental domain. This is an exercise in scalability as the task involves building and evaluating a robust system in a significantly larger domain than we have previously worked on. Table 1 summarizes the domains and the status of each project.

In this paper we will address architectural concerns in designing and building a generic dialogue shell. We will first describe the component-level architecture, which reflects our experience in building systems, with a special focus on the need to separate domain-independent aspects of the system from the domain-specific components that create a specific application domain. We then describe our programming level architecture, which has proven very effective in supporting system development and porting to new applications. Finally, we will describe a few specific components in the system to illustrate our approach of developing domain-independent components that can then be rapidly tailored to a specific domain.

2 A Generic Architecture for Dialogue Systems

Simple dialogue systems may consist of a fixed sequence of processing stages, starting with speech recognition, then parsing/analysis, dialogue management and response generation. As the dialogues to be handled become more complex, however, such a simple architecture does not work effectively. For instance, our current system shell under development has six separate modules that together provide the dialogue management: discourse context management, reference resolution, intention recognition, the behavioral agent, the plan manager, and response planning. Each of these plays a distinct role, and while they could be collapsed together for a particular application (as we did in the original TRAINS system), such monolithic modules are hard to construct and debug, and are difficult to modify for a new task and domain. Figure 1 shows the core set of modules in the generic dialogue shell, and Table 2 gives a brief description of each. The heavier arrows in Figure 1 show the main flow of processing from an input utterance to a response, and the lighter arrows indicate inter-component interactions along the way. In addition, all modules have access to a common semantic hierarchy and to a world KB manager that handles queries about the current situation, managing the interfaces to domain dependent reasoners and knowledge bases as needed.

One of the key things to note about this architecture is the separation of the basic dialogue system components from the more domain-specific components that provide the application (shown within the dotted lines at the lower left corner of Figure 1). To illustrate this separation, consider a specific example: a travel-agent application. The back-end would provide schedule and reservation information, booking, and so on, much as current computer systems provide to human travel agents. The Behavioral Agent and Plan Manager would be driven from a specification of desired behavior of the system as a travel agent, including the actions it typically will be asked to perform (*e.g.*, what information is relevant to the customer when planning a trip), what obligations it has (*e.g.*, find the customer the most convenient or cheapest flights), and a specification of how to perform actions (*e.g.*, to book a ticket, first get credit card information, then interact with the reservation system, confirm booking with user, and so on).

As another example, in the TRIPS-PACIFICA system, the back-end systems included a movement planner (choosing movement actions including the vehicles and cargoes involved), route planning (based on map and vehicle capabilities), and scheduling (based on nominal travel times). The Behavioral Agent knew the

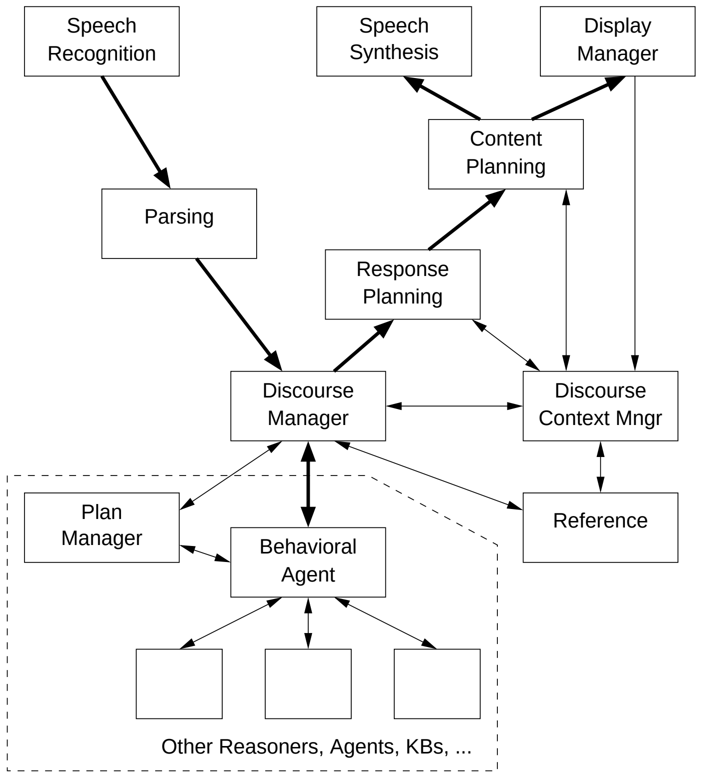

Fig. 1. Flow of information between TRIPS components

various plan-related operations a person might request, such as introducing a new goal, modifying an existing plan, evaluating a plan (*e.g.*, by asking "how long will that take?"), and so on. It also knew how to perform these actions as necessary and convey the resulting information back to the user. For example, to develop a plan for a new goal, it needed to first determine the movement actions required, then get the planner to incrementally modify the appropriate (usually current) plan to add the actions, call the router to find routes, and finally call the scheduler to produce a time-line. The resulting changes to the plan were used to produce a spoken and graphical response.

Separating the Behavioral Agent from the dialogue management is also key to supporting mixed-initiative interaction. The behavioral agent may have goals independent of the current conversation that might influence its response. For example, in an emergency management task, the behavioral agent may decide that it is more important to notify the user that an ambulance has become inoperative rather than to answer the user's current query about the weather forecast. Thus it might ignore the question temporarily (still retaining the obligation to answer later). Determining when to do this, of course, will be a domain-specific decision.

The Behavioral Agent also serves to encapsulate the domain-specific information,

Table 2. *Key modules in the abstract dialogue shell architecture*

Module	Function
Speech Recognition (SR)	Transforming speech input into a word stream or word lattice
Parser	Transforming the SR output into interpretations, each a set of conventional speech acts, using full and robust parsing techniques
Reference Manager (REF)	Identifying the most salient referents for referring expressions such as noun phrases
Discourse Context Manager	Maintaining the global (topic flow) and local (salience with a topic) discourse context
Discourse Manager (DM)	Identifying the intended speech act, current task, current step in the current task, and system obligations arising from the dialogue
Behavioral Agent (BA)	Determines system actions (*e.g.*, answer a question, notify of a problem, request clarification); Manages the interface to the back-end systems.
Plan Manager	Constructing, modifying, evaluating, and executing plans (whether they are the subject of the conversation or the task being executed)
World KB	Maintains a description of the current state of the world under differing assumptions (*e.g.*, based on different plans or hypotheses)
Response Planner	Determining the best communicative act(s) (and their content) to accomplish the system's current goals and discourse obligations
Content Planner	Determining how to realize the planned speech acts
Display Manager	Managing the visual presentations given the available displays

for the rest of the system interacts with the back-end systems only via the behavioral agent. By defining a generic interface to the behavioral agent in terms of plans, actions, goals, and so on, most of the system can be built independent of any specific domain. For example, The Discourse Manager (DM) coordinates a range of processes to recognize the user's intentions underlying the utterance and to compute new discourse obligations (*e.g.*, if asked a question, one should respond) (Traum & Allen 1994). The DM first receives input in the form of surface speech acts that are computed by the parser using the output from the speech recognition system. It is driven by a set of interpretation rules that match the surface speech acts and invoke modules such as the Reference Manager, Context Manager, Plan Manager and the Behavioral Agent to produce and/or evaluate possible interpretations. It might seem that the DM needs domain-specific information to perform its task. Instead, however, it deals with abstract intentions such as introducing a new goal, modifying an existing plan, and requesting background information. Intentions of this form are evaluated by the behavioral agent and plan manager with respect to the specific domain and the results are passed back to the DM. The DM does not have to know the details of the specific domain.

3 Program Level Infrastructure

We now turn to the second level of architecture, namely the programming infrastructure used to build dialogue systems. Portability to new domains and flexibility in accommodating new components requires a clear and explicit separation of responsibilities between the components. Not coincidentally, this is also good software engineering practice. Importantly for our team of researchers, it also allows work on individual components or small groups of components to proceed independently, with later integration into the complete system (possibly even at run-time).

3.1 Message-Passing Communication

Our program-level architecture consists of a set of loosely-coupled, heterogenous components that communicate by exchanging messages. Over the past several years, we have developed an extensive robust infrastructure for deploying components. Before describing this infrastructure, it is worth considering why we chose an explicit message-passing framework for inter-component communication rather than an alternative such as RPC or object-oriented method calls.

The first reason is that explicit message-passing provides platform- and language-independence. In our work, we use a variety of languages and platforms including Lisp, C, C++, Java, and Perl on Unix, Windows, and Macintosh platforms. TRIPS components can also communicate with other systems via agent frameworks such as the Open Agent Architecture (OAA) (Cohen *et al.* 1994) and object frameworks such as CORBA (OMG 1999). From the outset, we didn't want to restrict ourselves to a particular object framework or communication model.

The second reason for using explicit message-passing is the separation of transport and content information. The transport mechanisms and protocols are supported (and enforced) by the infrastructure, while the content is negotiated between the components. This negotiation can be at compile-time, in the case of objects, or at run-time in the case of agents. But the fact that message-passing separates transport and content decisions allows us to build a robust communication infrastructure without requiring prior universal agreement about content.

Third, and extremely important for us as researchers and system developers interested in rapid portability, is that explicit message-passing provides easy access to information needed for debugging. The message traffic in a run of the system is logged and can be used for post-mortem debugging and even replaying entire sessions.

3.2 Syntax and Semantics of Messages

In the TRIPS message-passing framework, components exchange messages using either the Knowledge Query and Manipulation Language (KQML) (Labrou & Finin 1997), designed as part of the DARPA Knowledge Sharing Effort, or the Agent Communication Language specified by the Foundation for Intelligent Physical Agents (FIPA 1999). These languages are quite similar—both use a Lisp-like s-expression syntax and are based loosely on the semantics of speech acts (*e.g.*, (Searle 1969)).

A message in these languages consists of a verb (or performative), indicating the speech act intended by the message, followed by a set of parameters specifying particulars of the message. Examples of KQML performatives include "TELL" and "ASK-IF". Messages can be addressed to other agents, and there is support in the specifications for connecting individual messages into conversational threads.

Importantly, neither KQML nor FIPA ACL specifies the semantics of the content of the messages. That is, there is a parameter with which the content of the message can be specified, but this parameter is not interpreted by the message-passing infrastructure. While this makes it difficult to specify and verify agents in terms of their communicative behaviors, it does make it possible to use these frameworks without solving a host of difficult problems in reasoning about other agents (Cohen & Levesque 1990). The fact that components developed using KQML or FIPA ACL share a syntax and a basic agreement about the meaning of the performatives generally makes their integration easier than if they communicated only via specialized interfaces.

3.3 The TRIPS Facilitator

The TRIPS message-passing infrastructure is based on a hub topology. The central node in the network of components is the TRIPS facilitator. The choice of a hub topology was a pragmatic one. While such an organization does lead to a possible single point of failure for the network if the hub goes down, in practice the advantages (such as support for validation and logging) are more important for a research system.

The main job of the Facilitator is to route each message to its intended receiver. In so doing, it performs full syntactic validation of the message with respect to the underlying protocol (*e.g.*, KQML). This again simplifies component development by removing much of the burden of protocol-level error-checking. To perform the routing, the TRIPS Facilitator allows clients to register one or more names. The Facilitator also allows the online specification of "client groups" to which individual clients can subscribe. In the TRIPS prototype system, these groups are used to describe categories of services, such as "user input" or "display." The Facilitator accepts group names (*i.e.*, service classes in the TRIPS prototype) as valid recipients for messages, and routes such messages to all members of the group.

These capabilities support an extended form of point-to-point addressing between components exchanging messages. However, in a system like TRIPS where the components are effectively members of an agent society, broadcasting is a common way of informing other agents of new information or requesting services. The TRIPS Facilitator therefore supports two separate forms of broadcast: (1) a “true” broadcast, which is used for various control messages, such as to indicate that a conversation is starting or has ended; and (2) a “selective broadcast”, a subsumption mechanism in which components can indicate interest in the output of a particular client or client group. Combined with the use of client groups, selective broadcast provides a powerful and flexible communication framework. For example, the Parser can indicate interest in the “user input” group, of which the Speech Recognizer is a member, along with other clients.

In summary, the TRIPS Facilitator provides robust message-passing facilities built on TCP sockets. It provides naming services, content- and service-based addressing, broadcast, selective broadcast, and logging. It has many similarities with agent-based communication frameworks like the OAA and can easily be extended to interact with OAA agents. We differ significantly from the Communicator Architecture (Goldschen & Loehr 1999) in that we do not use “Hub scripts” to control the flow of information between the components. We believe that the communication infrastructure should look after getting the messages where they need to go, and that the patterns of communication should be controlled by the components themselves. Programming these connections into the hub, while potentially useful for quick prototyping, eliminates the possibility of more sophisticated agents reasoning about their communicative behavior in order to use other agents and services more effectively.

4 Issues in Porting to New Applications

The final issue we will discuss is the issue of adapting the generic dialogue shell to new domains. As discussed above, our general strategy is to develop generic components that are domain-independent but limited to practical dialogues. In some cases, a generic component can work “as is” in a new domain. In others, we need techniques for rapidly specializing the components to perform effectively in each specific domain. We will discuss only a few specific examples here as there is not the space to cover every aspect of the system in detail.

4.1 Speech Recognition

We use a general-purpose speech recognizer, Sphinx-II (Huang *et al.* 1993) and generic acoustic and pronunciation models to achieve generality. We then tailor the lexicon and the language model to the task domain in order to achieve good recognition performance. This is especially important because our dialogues involve more spontaneous speech than applications such as dictation.

Most approaches to building language models for new domains are framed as adaptation problems: assume the availability of a good general model that can be made more specific by incorporating knowledge from a text corpus in a new domain. However, these approaches require the existence of a corpus for the new domain, which may become prohibitively lengthy and expensive. We have successfully built language models for new domains in a very short time by applying two techniques (Galescu, Ringger, & Allen 1998). First, in the absence of text data in the task domain, we generate an artificial corpus from a hand-coded context-free grammar (CFG) that is easily adaptable to new domains, and train a language model on this corpus. Second, we use a class-based approach to LM adaptation from out-of-domain corpora that allows us to re-use the dialogue data collected from other practical dialogue domains to build models for new domains.

The first technique involves generating an artificial corpus by Monte Carlo sampling from a hand-coded task-specific context-free grammar The purpose of the artificial corpus is to provide a source of plausible word collocations for the new domain. As such, a simple grammar based on a blend of syntactic and semantic categories is sufficient, and has the advantage of being very easy to write. Our grammars contain just a few hundred rules. We built the first CFG (for the Pacifica domain) starting from a few sentences (*e.g.*, from a script) in a manner similar to that suggested by Rayner (1997).

The second technique we use to obtain training material for language modeling is based on reusing data from other domains. Language models trained on the data from other domains might actually give worse performance than not using any language model at all. Therefore, we need to provide a transformation between language models in the remote domain and language models in the target domain. To do this we use a class-based approach. The general procedure for generating class-based n-gram models (Issar 1996) follows three steps: (1) tagging the corpus according to some predefined word-class mapping; (2) computing a back-off n-gram class model from the tagged text corpus; and (3) converting back to a word model using the word-class mappings in the class tag dictionary. We achieve the adaptation by using a tag dictionary specific to the remote domain for tagging the corpus (in step 1), and using a tag dictionary specific to the target domain to obtain a word-based LM from the class-based LM (in step 3). The success of this technique depends heavily on the compatibility between the two tag dictionaries being used. To achieve this kind of compatibility, we maintain a domain-independent tag dictionary (containing function words, pronouns, many common-use words and phrases of basic conversational English), and a specialized tag dictionary for each task domain. Thus, whenever we work on a new domain, we only need to adapt the specialized tag dictionary.

One disadvantage of this approach is that no statistics are available for words in the target domain associated with tags that don't appear in the remote corpus. We alleviate this problem by interpolating language models from several practical dialogue domain. After deploying the system, as text from the target domain becomes available, it can be used for building a domain-specific model. This may be used for further adaptation of the initial model (Rudnicky 1995), or, if reliable enough, it may replace the initial model.

We conducted a thorough evaluation on the Pacifica domain; see (Galescu, Ringger, & Allen 1998) for extensive details. We observed good perplexity and word error rate results for both techniques, and especially for their combination. As a consequence, we have used the above techniques for building initial language models in all domains to which we have ported the TRIPS system (see Table 1).

4.2 Parsing

Working with real-time dialogue places many demands on the parser. If an analysis can be found, this must be done quickly so that the user does not have to wait too long. At the same time, we want the grammar to cover a wide range of grammatical constructs, but extending the grammatical coverage often decreases the efficiency of the parser. We address this problem by extensive use of selectional (*i.e.*, semantic) restrictions. While it has been argued that selectional restrictions are problematic as a general theory of semantics, we have found them to be practical and effective for specific domains. The challenge is to keep a general domain-independent grammar and lexicon for portability and then adapt it to a new domain for accuracy and efficiency. The parser uses a chart-based best-first algorithm that accepts input incrementally. The grammar is a fairly standard feature-based context-free formalism. The parser and grammatical formalism are based on that in (Allen 1994).

We start with a grammar and a core lexicon that describes general English usage in practical dialogues and then obtain a domain-specific parser by compiling in a domain semantics. For example, there are general patterns for the syntax of a verb phrase: a verb can only take a certain number of complements that are well-defined constituents. Princeton WordNet (Miller 1995) identifies 35 verb frames encountered in English. While some domains may not involve all these frames, it is reasonable to assume that the number and syntactic types of verb arguments is domain independent. Therefore, they are part of the core lexicon and grammar. Similarly, there are semantic properties that stay the same in all domains. For example, a car is a physical object in all domains, and it cannot conceivably be a living entity.

Specific domains differ primarily in the semantics they assign to lexical items. Not only do words have different meanings in different domains, but the notion of a relevant semantic property varies. For example, in the Pacifica domain it is important to distinguish fixed objects such as cities from movable objects such as people and cargo. On the other hand, in a kitchen design system we will need to distinguish between furniture and appliances, something that is not at all important in Pacifica, where they will be all treated as cargo. In a similar vein, the verb "move" in Pacifica refers to a transportation action, whereas in the kitchen design domain it refers to a change in the kitchen plan, and transportation actions are unknown. In Pacifica, a window in a house would be fixed (not movable), whereas in the kitchen design domain it is movable (in the design plan).

To build a domain specific grammar, we define mappings from the domain-independent representations to domain-specific predicates. For example, the generic grammar contains a domain-independent verb class "MOVE". In the Pacifica domain, transportation actions are important and defined by a domain-specific predicate TRANSPORT. As a result, we produce domain-specific semantic restrictions on the roles for MOVE-class verbs. For example, the *Instrument* is required to be a vehicle, the *Theme* is restricted to be movable (also a domain-dependent feature that selects the set of objects that can be moved in the domain), and so on. We have now significantly reduced the number of sentences that could be parsed, which increases the efficiency of the parser, improves structural disambiguation, and provides strong semantic guidance for robust processing to recover from speech recognition errors. In the kitchen domain, on the other hand, MOVE would map to a kitchen plan modification action, where the *Agent* can only be one of the dialogue participants, the *Theme* is a "movable" design element in the design plan, while the *Instrument* will be always absent. The grammars that result from those two specializations result in very different sentences being understandable, giving us the advantage of domain-specific semantic grammars without the complexity of having to define different grammar rules by hand.

4.3 Reference Resolution

Resolution of referring expressions is part of the semantic interpretation phase of the Dialogue Manager. In TRIPS, referring expressions are resolved by first constructing a list of known properties of the referent, calling knowledge managers within the system (such as the context manager and display manager) to return entities matching those properties, then calculating a confidence rating that each matching entity is the correct referent. By depending on other knowledge managers for any domain-specific information or reasoning, the reference resolution module can remain effectively domain-independent in generating possible candidates. Calculation of the confidence rating varies depending on the form of the referring expression (*e.g.*, pronoun, definite or indefinite description) and is domain independent. Returning a list of possible referents, rather than just the one most probable referent, leaves the Dialogue Manager with more flexibility to combine the results with information from other sources such as domain-specific reasoners. Because there may be different sources of information available in different domains, reference resolution must be robust to missing or incomplete information.

In practical dialogues, we find a larger variety of referring expressions and referring behavior than one finds in other language genres. We analyzed two spoken corpora for their referential behavior (Byron 1999). While 89% of the pronouns in a spoken monologue corpus were co-referential with a noun phrase, only 25% of the pronouns in our practical dialogue corpus were. That means that using a pronoun resolution technique that relies solely on identifying noun phrase antecedents in the prior discourse (a commonly used technique in language understanding systems) is entirely inadequate for practical dialogues. In this genre, reference to events, propositions, entire stretches of discourse, etc. is much more common. Therefore, the pronoun resolution technique used must allow a wide variety of candidate referents. Practical dialogues also contain many more bare demonstrative pronouns. In the TRAINS corpus (Heeman & Allen 1995), 50% of the pronouns are demonstrative, compared to less than 10% in the spoken monologue corpus.

To resolve this variety of referring expressions, reference resolution often needs to draw on semantic type information. For example, verb semantic restrictions computed by the parser can be used to limit the possible semantic type of allowable referents. To do this we use a semantic hierarchy that is shared by all modules in the system and that contains a combination of domain-independent and domain-dependent information. Reference resolution does not contain details about the representation of domain objects, it only knows which knowledge manager to query to get particular information. For example, to resolve a reference to "the green route," the reference module does not need to know what routes are or how they are represented in this domain. It simply handles ROUTE as a possible object type and knows to ask the Display Manager which ones are currently green. The flexibility of our message-passing infrastructure supports this style of component interactions without hard-coding any domain-specific information into the Reference component.

4.4 Content planning and generation

In the current TRIPS system, the response planner receives conversational goals from the discourse manager (which is driven by directions from the behavioral agent). It selects content for output and passes the generator a set of role-based logical forms with associated speech acts. The generator decides which logical forms to produce and how to sequence them, and then passes commands on to modality-specific generators (prosodically-annotated textual sentences are sent to the speech synthesizer; commands to display maps and charts are sent to the Display Manager).

Our current language generator uses a domain-specific template-based approach. Templates work well in near-fixed-initiative situations where there are few types of system utterances; they are fast and flexible. Grammar-based approaches work well when broad language coverage is needed, but are not usually fast enough for use in real-time dialogue systems.

The goal of our current work is to develop a conversational agent capable of generating varied and complex content in a mixed-initiative fashion. Accordingly, our proposed generation framework differs significantly from our existing one. It is based on theoretical and empirical understandings of what dialogue contributions involve (Traum & Hinkelman 1992) and is designed to support incremental, real-time generation; broad language coverage; and flexibility in modifying system behavior and switching to new domains. We consider the three stages of generation (planning intentions, planning content and planning form) as three different aspects of utterances that have to be planned, possibly simultaneously (Reithinger 1991; De Smedt, Horacek, & Zock 1996). Grounding and turn-taking acts, for instance, involve the planning of intentions only. Also, these acts often begin a turn, so generating these acts quickly can give a conversational agent time to produce other acts that may involve more processing. Because the form of such acts can be selected from a set of conventional forms, a template-based approach can both be domain-independent and be fast enough for real-time interaction.

Those utterances that speakers produce to fulfill intentions arising directly from the domain or the task being solved often have content that must be expressed. The form may or may not be important. We will generate these types of utterances using templates when the forms are limited, but use grammar-based generation in the more complex cases. Other utterances are produced primarily to complete an argumentation act. Their production involves the planning of intentions, semantic content and surface form, and they would usually need to be generated using a grammar.

Our proposed generator architecture is described in detail in (Stent 1999). The new response planner will plan the system's intentions for the continuing dialogue by considering a set of possible interpretations from the DM and the Behavioral Agent. It must ensure that none of the selected goals conflict or are redundant. Generation goals are represented as sets of intention by content pairs. The generator maintains a list of these sets (we call this list the intention-set). Each set is sent to all the generation modules. The output from each module is a surface form and a set of intentions fulfilled by that form. A gate-keeper at the end removes intentions from the intention-set as they are fulfilled. It can also add sets of intentions to the intention-set, for instance to keep the turn or if the agent is interrupted. Finally, it can minimize over-generation by selecting which results to produce, if it gets simultaneous results that satisfy the same intentions.

Consider an example. Imagine a user has just made a statement to the agent. The agent wants to acknowledge part of the statement (grounding) and ask a question about another part. So the intention-set looks like: {*take-turn, acknowledge(Utt1), info-request(Content)*} (items with initial capital letters are variables). This set gets passed to all modules. The turn-taking module returns "Uh" for *take-turn* and the grounding module returns "Okay" for *take-turn* and *acknowledge(Utt1)*. The gate-keeper therefore removes *take-turn* and *acknowledge(Utt1)* and produces "Okay". If a pause of more than, say, half-a-second ensues, the gate-keeper might add the intention *keep-turn* to the intention-set which will feed it to the various modules. However, happily the gate-keeper quickly receives a result for *info-request(Content)* which it produces, removing that intention (and therefore the whole set) from the intention-set.

Real-time generation is very important in the context of dialogue. We believe our architecture will allow for real-time responses with grounding and turn-taking acts, "buying" the system more time to plan utterances that convey complex content.

5 Conclusions

We have argued that building a generic shell for dialogue systems is both a valuable and feasible research goal. Such a shell would provide generic language understanding and dialogue management functions that could then be adapted relatively easily to new task domains. We believe that useful dialogue systems in future applications must go beyond simple menu-oriented interaction and deal with complex problems such as intention recognition, topic tracking, grounding, reference resolution and incremental response planning. To address these goals we have proposed an architecture derived from our experience building dialogue systems in multiple domains. While still very much a work in progress, we believe that we have laid a strong foundation for rapid progress in the next few years. Our agent-oriented architecture based on KQML communication supports a flexible, plug-and-play approach to system construction. It is compatible with ongoing work in the emerging area of agent-based systems, where we believe many of the new applications of dialogue systems will arise. Our component-level architecture identifies the key capabilities needed for next-generation dialogue systems that will have to handle more complex tasks than those currently studied. Our approach of making a clean separation between domain-independent and domain-specific information, and investigating techniques for rapid adaptation of the domain-independent components provides us with the necessary compromise between the generality required for portability and the efficiency required for practical applications.

References

Allen, J. F.; Schubert, L. K.; Ferguson, G.; Heeman, P.; Hwang, C. H.; Kato, T.; Light, M.; Martin, N. G.; Miller, B. W.; Poesio, M.; and Traum, D. R. 1995. The TRAINS project: A case study in defining a conversational planning agent. *Journal of Experimental and Theoretical AI* 7:7–48.

Allen, J. F. 1994. *Natural Language Understanding, 2nd ed.* Menlo Park, CA: Benjamin/Cummings Publishing Co.

Byron, D. K. 1999. Analysis of pronominal reference in two spoken language collections: TRAINS-93 spontaneous task-oriented dialogue and boston university radio news prepared monologue. Technical Report 703, Department of Computer Science, University of Rochester, Rochester, NY.

Cohen, P., and Levesque, H. 1990. Intention is choice with commitment. *Artifical Intelligence* 42:213–261.

Cohen, P. R.; Cheyer, A. J.; Wang, M.; and Baeg, S. C. 1994. An open agent architecture. In *AAAI Spring Symposium on Software Agents*, 1–8.

De Smedt, K.; Horacek, H.; and Zock, M. 1996. Architectures for natural language generation: Problems and perspectives. In *Trends in Natural Language Generation: An Artificial Intelligence Perspective.* Berlin, Germany: Springer-Verlag. 17–46.

Ferguson, G., and Allen, J. F. 1998. TRIPS: An integrated intelligent problem-solving assistant. In *Proceedings of the Fifteenth National Conference on AI (AAAI-98)*, 567–573.

Ferguson, G.; Allen, J. F.; Miller, B. W.; and Ringger, E. K. 1996. The design and implementation of the TRAINS-96 system: A prototype mixed-initiative planning assistant. TRAINS Technical Note 96-5, Department of Computer Science, University of Rochester, Rochester, NY.

FIPA. 1999. Agent communication language. FIPA Spec 2–1999 (draft version 0.2), Foundation for Intelligent Physical Agents, Geneva, Switzerland.

Galescu, L.; Ringger, E. K.; and Allen, J. F. 1998. Rapid language model development for new task domains. In *Proceedings of the ELRA First International Conference on Language Resources and Evaluation (LREC'98).*

Goldschen, A., and Loehr, D. 1999. The role of the DARPA communicator architecture as a human computer interface for distributed simulations. Technical report, MITRE Corporation.

Heeman, P. A., and Allen, J. F. 1995. The TRAINS-93 dialogues. TRAINS Technical Note 94-2, Department of Computer Science, University of Rochester, Rochester, NY.

Huang, X.; Alleva, F.; Hon, H.-W.; Hwang, M.-Y.; Lee, K.-F.; and Rosenfeld, R. 1993. The SPHINX-II speech recognition system: An overview. *Computer Speech and Language* 7(2):137–148.

Issar, S. 1996. Estimation of language models for new spoken language applications. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP-96)*, 869–872.

Labrou, Y., and Finin, T. 1997. A proposal for a new KQML specification. Technical Report CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD.

Miller, G. 1995. WordNet: A lexical database for english. *Communications of the ACM* 38(5):39–41.

OMG. 1999. *The Common Object Request Broker: Architecture and Specification (Revision 2.3)*. Object Management Group.

Rayner, M., and Carter, D. 1997. Hybrid language processing in the spoken language translator. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP-97)*, 107–110.

Reithinger, N. 1991. POPEL – a parallel and incremental natural language generation system. In Paris, C. L.; Swartout, W. R.; and Mann, W. C., eds., *Natural Language Generation in Artificial Intelligence and Computational Linguistics*. Boston, MA: Kluwer Academic Publishers. 179–199.

Rudnicky, A. I. 1995. Language modeling with limited domain data. In *Proceedings of the ARPA Spoken Language Technology Workshop*.

Searle, J. R. 1969. *Speech Acts: An essay in the philosophy of language*. Cambridge, England: Cambridge University Press.

Stent, A. J., and Allen, J. F. 1997. TRAINS-96 system evaluation. TRAINS Technical Note 97-1, Department of Computer Science, University of Rochester, Rochester, NY.

Stent, A. 1999. Content planning and generation in continuous-speech spoken dialog systems. In *Proceedings of the KI-99 workshop "May I Speak Freely?"*.

Traum, D. R., and Allen, J. F. 1994. Discourse obligations in dialogue processing. In *Proceedings of the Thirty-second Annual Meeting of the Association for Computational Linguistics (ACL-94)*.

Traum, D., and Hinkelman, E. 1992. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence* 8(3):575–599.