Error Handling in the RavenClaw Dialog Management Framework

Dan Bohus
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, 15213
dbohus@cs.cmu.edu

Alexander I. Rudnicky
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, 15213
air@cs.cmu.edu

Abstract

We describe the error handling architecture underlying the RavenClaw dialog management framework. The architecture provides a robust basis for current and future research in error detection and recovery. Several objectives were pursued in its development: task-independence, ease-of-use, adaptability and scalability. We describe the key aspects of architectural design which confer these properties, and discuss the deployment of this architecture in a number of spoken dialog systems spanning several domains and interaction types. Finally, we outline current research projects supported by this architecture.

1 Introduction

Over the last decade, improvements in speech recognition and other component technologies have paved the way for the emergence of complex task-oriented spoken dialog systems. While traditionally the research community has focused on building information-access and command-and-control systems, recent efforts aim towards building more sophisticated language-enabled agents, such as personal assistants, interactive tutors, open-domain question answering systems, etc. At the other end of the complexity spectrum, simpler systems have already transitioned into day-to-day use and are becoming the norm in the phone-based customer-service industry.

Nevertheless, a number of problems remain in need of better solutions. One of the most important limitations in today's spoken language interfaces is their lack of robustness when faced with understanding errors. This problem appears across all domains and interaction types, and stems primarily from the inherent unreliability of the speech recognition process. The recognition difficulties are further exacerbated by the conditions under which these systems typically operate: spontaneous speech, large vocabularies and user populations, and large variability in input line quality. In these settings, average word-error-rates of 20-30% (and up to 50% for non-native speakers) are quite common.

Left unchecked, speech recognition errors can lead to two types of problems in a spoken dialog system: *misunderstandings* and *non-understandings*. In a *misunderstanding*, the system obtains an incorrect semantic interpretation of the user's turn. In the absence of robust mechanisms for assessing the reliability of the decoded inputs, the system will take the misunderstanding as fact and will act based on invalid information. In contrast, in a *non-understanding* the system fails to obtain an interpretation of the input altogether. Although no false information is incorporated in this case, the situation is not much better: without an appropriate set of recovery strategies and a mechanism for diagnosing the problem, the system's follow-up options are limited and uninformed. In general, unless mitigated by accurate error awareness and robust recovery mechanisms, speech recognition errors exert a strong negative impact on the quality and ultimately on the success of the interactions (Sanders et al, 2002).

Two pathways towards increased robustness can be easily envisioned. One is to improve the accuracy of the speech recognition process. The second is to create mechanisms for detecting and gracefully handling potential errors at the conversation level. Clearly, these two approaches do not

Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP), pages 225–232, Vancouver, October 2005. ©2005 Association for Computational Linguistics

stand in opposition and a combined effort would lead to the best results. The error handling architecture we describe in this paper embodies the second approach: it aims to provide the mechanisms for robust error handling at the dialog management level of a spoken dialog system.

The idea of handling errors through conversation has already received a large amount of attention from the research community. On the theoretical side, several models of grounding in communication have been proposed (Clark and Schaefer, 1989; Traum, 1998). While these models provide useful insights into the grounding process as it happens in human-human communication, they lack the decision-making aspects required to drive the interaction in a real-life spoken dialog system. In the Conversational Architectures project, Paek and Horvitz (2000) address this challenge by developing a computational implementation of the grounding process using Bayesian belief networks. However, questions still remain: the structure and parameters of the belief networks are handcrafted, as are the utilities for the various grounding actions; due to scalability and task-representation issues, it is not known yet how the proposed approach would transfer and scale to other domains.

Three ingredients are required for robust error handling: (1) the ability to detect the errors, (2) a set of error recovery strategies, and (3) a mechanism for engaging these strategies at the appropriate time. For some of these issues, various solutions have emerged in the community. For instance, systems generally rely on recognition confidence scores to detect potential misunderstandings (e.g. Krahmer et al., 1999; Walker et al., 2000) and use explicit and implicit confirmation strategies for recovery. The decision to engage these strategies is typically based on comparing the confidence score against manually preset thresholds (e.g. Kawahara and Komatani, 2000). For non-understandings, detection is less of a problem (systems know by definition when non-understandings occur). Strategies such as asking the user to repeat or rephrase, providing help, are usually engaged via simple heuristic rules.

At the same time, a number of issues remain unsolved: can we endow systems with better error awareness by integrating existing confidence annotation schemes with correction detection mechanisms? Can we diagnose the non-understanding errors on-line? What are the tradeoffs between the various non-understanding recovery strategies? Can we construct a richer set of such strategies? Can we build systems which automatically tune their error handling behaviors to the characteristics of the domains in which they operate?

We have recently engaged in a research program aimed at addressing such issues. More generally, our goal is to develop a task-independent, easy-to-use, adaptive and scalable approach for error handling in task-oriented spoken dialog systems. As a first step in this program, we have developed a modular error handling architecture, within the larger confines of the RavenClaw dialog management framework (Bohus and Rudnicky, 2003). The proposed architecture provides the infrastructure for our current and future research on error handling. In this paper we describe the proposed architecture and discuss the key aspects of architectural design which confer the desired properties. Subsequently, we discuss the deployment of this architecture in a number of spoken dialog systems which operate across different domains and interaction types, and we outline current research projects supported by the proposed architecture.

2 RavenClaw Dialog Management

We begin with a brief overview of the RavenClaw dialog management framework, as it provides the larger context for the error handling architecture.

RavenClaw is a dialog management framework for task-oriented spoken dialog systems. To date, it has been used to construct a large number of systems spanning multiple domains and interaction types (Bohus and Rudnicky, 2003): information access (RoomLine, the Let's Go Bus Information System), guidance through procedures (LARRI), command-and-control (TeamTalk), taskable agents (Vera). Together with these systems, RavenClaw provides the larger context as well as a test-bed for evaluating the proposed error handling architecture. More generally, RavenClaw provides a robust basis for research in various other aspects of dialog management, such as learning at the task and discourse levels, multi-participant dialog, timing and turn-taking, etc.

A key characteristic of the RavenClaw framework is the separation it enforces between the domain-specific and domain-independent aspects of dialog control. The domain-specific dialog control logic is described by a *Dialog Task Specification*,

essentially a hierarchical dialog plan provided by the system author. A fixed, domain-independent *Dialog Engine* manages the conversation by executing the given Dialog Task Specification. In the process, the Dialog Engine also contributes a set of domain-independent conversational skills, such as error handling (discussed extensively in Section 4), timing and turn-taking, etc. The system authoring effort is therefore minimized and focused entirely on the domain-specific aspects of dialog control.

2.1 The Dialog Task Specification

A Dialog Task Specification consists of a tree of *dialog agents,* where each agent manages a subpart of the interaction. Figure 1 illustrates a portion of the dialog task specification from RoomLine, a spoken dialog system which can assist users in making conference room reservations. The root node subsumes several children: Welcome, which produces an introductory prompt, GetQuery which obtains the time and room constraints from the user, DoQuery which performs the database query, and DiscussResults which handles the follow-up negotiation dialog. Going one level deeper in the tree, GetQuery contains GetDate which requests the date for the reservation, GetStartTime and GetEndTime which request the times, and so on. This type of hierarchical task representation has a number of advantages: it scales up gracefully, it can be dynamically extended at runtime, and it implicitly captures a notion of context in dialog.

The agents located at the leaves of the tree are called *basic dialog agents*, and each of them implements an atomic dialog action (dialog move). There are four types of basic dialog agents: ***I****nform* – conveys information to the user (e.g. Welcome), ***R****equest* – asks a question and expects an answer (e.g. GetDate), ***E****xpect* – expects information without explicitly asking for it, and ***EX****ecute* – implements a domain specific operation (e.g. DoQuery). The agents located at non-terminal positions in the tree are called *dialog agencies* (e.g. RoomLine, GetQuery). Their role is to plan for and control the execution of their sub-agents. For each agent in the tree, the system author may specify preconditions, completion criteria, effects and triggers; various other functional aspects of the dialog agents (e.g. state-specific language models for request-agents, help-prompts) are controlled through parameters.

The information the system acquires and manipulates in conversation is captured in *concepts*, associated with various agents in the tree (e.g. date, start_time). Each concept maintains a history of previous values, information about current candidate hypotheses and their associated confidence scores, information about when the concept was last updated, as well as an extended set of flags which describe whether or not the concept has been conveyed to the user, whether or not the concept has been grounded, etc. This rich representation provides the necessary support for concept-level error handling.

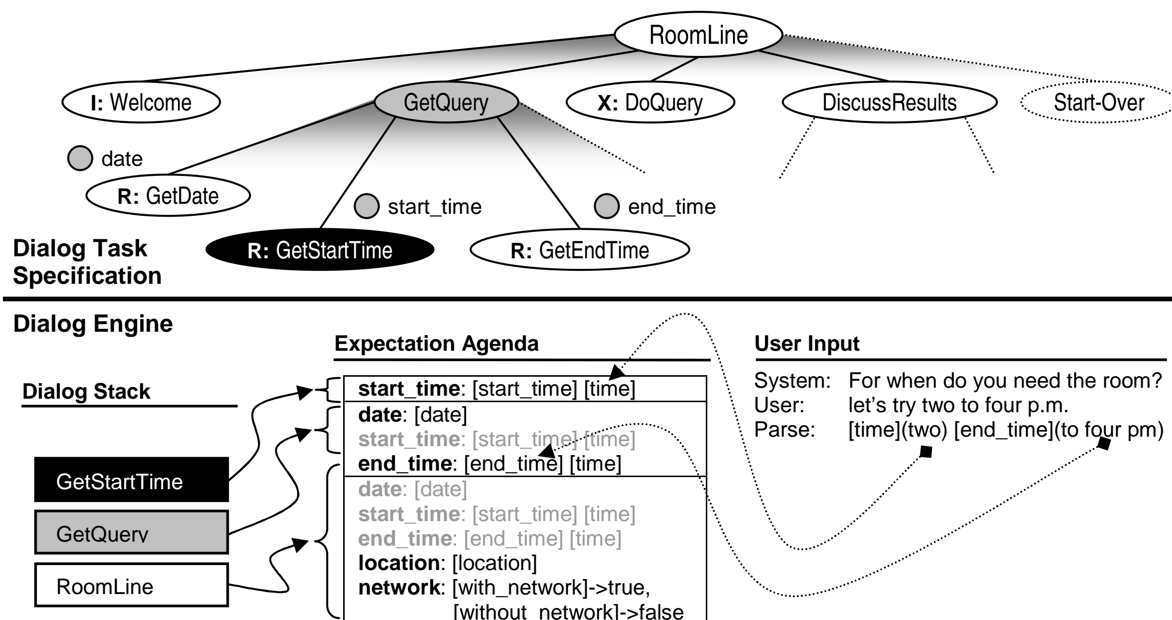

Figure 1: RavenClaw architecture

2.2 The Dialog Engine

The Dialog Engine is the core domain-independent component which manages the interaction by executing a given Dialog Task Specification. The control algorithms are centered on two data-structures: a dialog stack, which captures the dialog structure at runtime, and an expectation agenda, which captures the system's expectations for the user input at each turn in the dialog. The dialog is controlled by interleaving *Execution Phases* with *Input Phases*.

During an Execution Phase, dialog agents from the tree are placed on, and executed from the dialog stack. At the beginning of the dialog, the root agent is placed on the stack. Subsequently, the engine repeatedly takes the agent on the top of the stack and executes it. When dialog agencies are executed, they typically schedule one of their subagents for execution by placing it on the stack. The dialog stack will therefore track the nested structure of the dialog at runtime. Ultimately, the execution of the basic dialog agents on the leaves of the tree generates the system's responses and actions.

During an Input Phase, the system assembles the expectation agenda, which captures what the system expects to hear from the user in a given turn. The agenda subsequently mediates the transfer of semantic information from the user's input into the various concepts in the task tree. For the interested reader, these mechanisms are described in more detail in (Bohus and Rudnicky, 2003)

Additionally, the Dialog Engine automatically provides a number of conversational strategies, such as the ability to handle various requests for help, repeating the last utterance, suspending and resuming the dialog, starting over, reestablishing the context, etc. These strategies are implemented as *library dialog agencies.* Their corresponding sub-trees are automatically added to the Dialog Task Specification provided by the system author (e.g. the Start-Over agency in Figure 1.) The automatic availability of these strategies lessens development efforts and ensures a certain uniformity of behavior both within and across tasks.

3 The Error Handling Architecture

The error handling architecture in the RavenClaw dialog management framework subsumes two main components: (1) a set of *error handling strategies* (e.g. explicit and implicit confirmation, asking the user to repeat, etc.) and (2) an *error handling process* which engages these strategies.

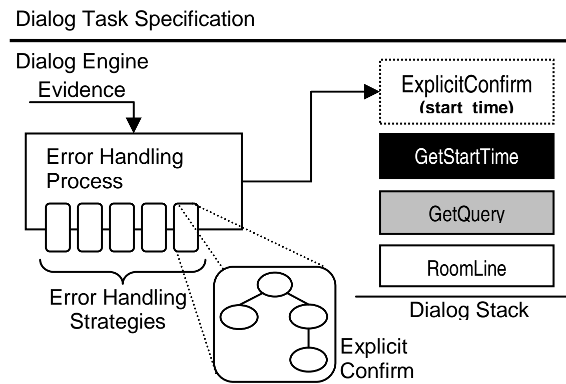

Figure 2: Error Handling – Block Diagram

The error handling strategies are implemented as library dialog agents. The decision process which engages these strategies is part of the Dialog Engine. This design, in which both the strategies and the decision process are decoupled from the dialog task, as well as from each other, provides a number of advantages. First, it ensures that the error handling mechanisms are reusable across different dialog systems. Second, the approach guarantees a certain uniformity and consistency in error handling behaviors both within and across systems. Third, as new error handling strategies are developed, they can be easily plugged into any existing system. Last, but not least, the approach significantly lessens the system authoring effort by allowing developers to focus exclusively on describing the dialog control logic.

The responsibility for handling potential understanding errors[1] is delegated to the Error Handling Process which runs in the Dialog Engine (see Figure 2). At each system turn, this process collects evidence and makes a decision with respect to engaging any of the error handling strategies. When necessary, it will insert an error handling strategy on the dialog stack (e.g. the ExplicitConfirm (start_time) strategy in Figure 2), thus modifying on-the-fly the task originally specified by the system author. The strategy executes and, once completed, it is removed from the stack and the dialog resumes from where it was left off.

[1] Note that the proposed framework aims to handle understanding errors. The corresponding strategies are generic and can be applied in any domain. Treatment of domain or task-specific errors (e.g. database access error, etc) still needs to be implemented as part of the dialog task specification.

3.1 Error Handling Strategies

The error handling strategies can be divided into two groups: strategies for handling potential misunderstandings and strategies for handling non-understandings.

For handling potential misunderstandings, three strategies are currently available: *Explicit Confirmation*, *Implicit Confirmation* and *Rejection*.

For non-understandings, a larger number of error recovery strategies are currently available: *AskRepeat* – the system asks the user to repeat; *AskRephrase* – the system asks the user to rephrase; *Reprompt* – the system repeats the previous prompt; *DetailedReprompt* – the system repeats a more verbose version of the previous prompt, *Notify* – the system simply notifies the user that a non-understanding has occurred; *Yield* – the system remains silent, and thus implicitly notifies the user that a non-understanding has occurred; *MoveOn* – the system tries to advance the task by giving up on the current question and moving on with an alternative dialog plan (note that this strategy is only available at certain points in the dialog); *YouCanSay* – the system gives an example of what the user could say at this point in the dialog; *FullHelp* – the system provides a longer help message which includes an explanation of the current state of the system, as well as what the user could say at this point. An in-depth analysis of these strategies and their relative tradeoffs is available in (Bohus and Rudnicky, 2005a). Several sample dialogs illustrating these strategies are available on-line (RoomLine, 2003).

3.2 Error Handling Process

The error handling decision process is implemented in a distributed fashion, as a collection of local decision processes. The Dialog Engine automatically associates a local error handling process with each concept, and with each request agent in the dialog task tree, as illustrated in Figure 3. The error handling processes running on individual concepts are in charge of recovering from misunderstandings on those concepts. The error handling processes running on individual request agents are in charge or recovering from non-understandings on the corresponding requests.

At every system turn, each concept- and request-agent error handling process computes and forwards its decision to a gating mechanism, which queues up the actions (if necessary) and executes them one at a time. For instance, in the example in Figure 3, the error handling decision process for the start_time concept decides to engage an explicit confirmation on that concept, while the other decision processes do not take any action. In this case the gating mechanism creates a new instance of an explicit confirmation agency, passes it the pointer to the concept to be confirmed (start_time), and places it on the dialog stack. On completion, the strategy updates the confidence score of the confirmed hypothesis in light of the user response, and the dialog resumes from where it was left off.

The specific implementation of the local decision processes constitutes an active research issue. Currently, they are modeled as Markov Decision Processes (MDP). The error handling processes running on individual concepts (*concept-MDPs* in

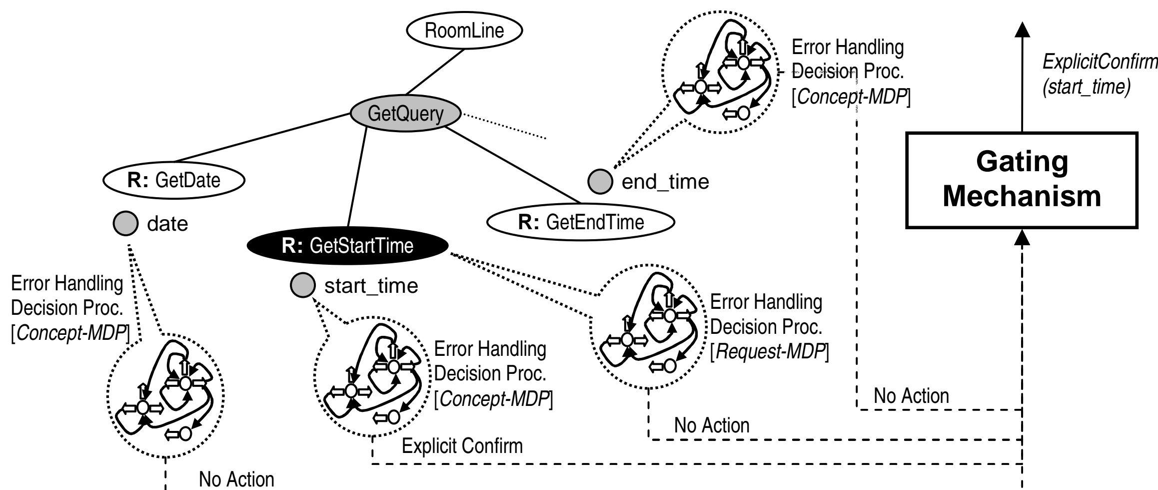

Figure 3: A Distributed Error Handling Process

Figure 3) are partially-observable MDPs, with 3 underlying hidden states: *correct*, *incorrect* and *empty*. The belief state is constructed at each time step from the confidence score of the top-hypothesis for the concept. For instance, if the top hypothesis for the start_time concept is 10 a.m. with confidence 0.76, then the belief state for the POMDP corresponding to this concept is: {P(correct)=0.76, P(incorrect)=0.24, P(empty)=0}. The action-space for these models contains the three error recovery strategies for handling potential misunderstandings, and *no-action*. The third ingredient in the model is the *policy*. A policy defines which action the system should take in each state, and is indirectly described by specifying the utility of each strategy in each state. Currently, a number of predefined policies (e.g. always-explicit-confirm, pessimistic, and optimistic) are available in the framework. Alternatively, system authors can specify and use their own policies.

The error handling processes running on request agents (*request-MDPs* in Figure 3) are in charge of handling non-understandings on those requests. Currently, two types of models are available for this purpose. The simplest model has three states: *non-understanding*, *understanding* and *inactive*. A second model also includes information about the number of consecutive non-understandings that have already happened. In the future, we plan to identify more features which carry useful information about the likelihood of success of individual recovery strategies and use them to create more complex models. The action-space is defined by the set of non-understanding recovery strategies presented in the previous subsection, and *no-action*. Similar to the concept-MDPs, a number of default policies are available; alternatively, system authors can specify their own policy for engaging the strategies.

While the MDP implementation allows us to encode various expert-designed policies, our ultimate goal is to learn such policies from collected data using reinforcement learning. Reinforcement learning has been previously used to derive dialog control policies in systems operating with small tasks (Scheffler and Young, 2002; Singh et al, 2000). The approaches proposed to date suffer however from one important shortcoming, which has so far prevented their use in large, practical spoken dialog systems. The problem is lack of scalability: the size of the state space grows very fast with the size of the dialog task, and this renders the approach unfeasible in complex domains. A second important limitation of reinforcement learning techniques proposed to date is that the learned policies cannot be reused across tasks. For each new system, a new MDP has to be constructed, new data has to be collected, and a new training phase is necessary. This requires a significant amount of expertise and effort from the system author.

We believe that the error handling architecture we have described addresses these issues in several ways. The central idea behind the distributed nature of the approach is to keep the learning problem tractable by leveraging independence relationships between different parts of the dialog. First, the state and action-spaces can be maintained relatively small since we are only focusing on making error handling decisions (as opposed to other dialog control decisions). A more complex task translates into a larger number of MDP instantiations rather than a more complex model structure. Second, both the model structure and parameters (i.e. the transition probabilities) can be tied across models: for instance the MDP for grounding the start_time concept can be identical to the one for grounding the end_time concept; all models for grounding Yes/No concepts could be tied together, etc. Model tying has the potential to greatly improve scalability since data is polled together and the total number of model parameters to be learned grows sub-linearly with the size of the task. Third, since the individual MDPs are decoupled from the actual system task, the policies learned in a particular system can potentially be reused in other systems (e.g. we expect that grounding yes/no concepts functions similarly at different locations in the dialog, and across domains). Last but not least, the approach can easily accommodate dynamic task generation. In traditional reinforcement learning approaches the state and action-spaces of the underlying MDP are task-specific. The task therefore has to be fixed, known in advance: for instance the slots that the system queries the user about (in a slot-filling system) are fixed. In contrast, in the RavenClaw architecture, the dialog task tree (e.g. the dialog plan) can be dynamically expanded at runtime with new questions and concepts, and the corresponding request- and concept-MDPs are automatically created by the Dialog Engine.

4 Deployment and Current Research

While a quantitative evaluation of design characteristics such as task-independence, scalability, and ease-of-use is hard to perform, a first-order empirical evaluation of the proposed error handling architecture can be accomplished by using it in different systems and monitoring the system authoring process and the system's operation.

To date, the architecture has been successfully deployed in three different spoken dialog systems. A first system, RoomLine (2003), is a phone-based mixed-initiative system that assists users in making conference room reservations on campus. A second system, the Let's Go! Bus Information System (Raux et al, 2005), provides information about bus routes and schedules in the greater Pittsburgh area (the system is available to the larger public). Finally, Vera is a phone-based taskable agent that can be instructed to deliver messages to a third party, make wake-up calls, etc. Vera actually consists of two dialog systems, one which handles incoming requests (Vera In) and one which performs message delivery (Vera Out). In each of these systems, the authoring effort with respect to error handling consisted of: (1) specifying which models and policies should be used for the concepts and request-agents in the dialog task tree, and (2) writing the language generation prompts for explicit and implicit confirmations for each concept.

Even though the first two systems operate in similar domains (information access), they have very different user populations: students and faculty on campus in the first case versus the entire Pittsburgh community in the second case. As a result, the two systems were configured with different error handling strategies and policies (see Table 1). RoomLine uses explicit and implicit confirmations with an optimistic policy to handle potential misunderstandings. In contrast, the Let's Go Public Bus Information System always uses explicit confirmations, in an effort to increase robustness (at the expense of potentially longer dialogs). For non-understandings, RoomLine uses the full set of non-understanding recovery strategies presented in section 3.1. The Let's Go Bus Information system uses the *YouCanSay* and *FullHelp* strategies. Additionally a new *GoToAQuieterPlace* strategy was developed for this system (and is now available for use into any other RavenClaw-based system). This last strategy asks the user to move to a quieter place, and was prompted by the observation that a large number of users were calling the system from noisy environments.

While the first two systems were developed by authors who had good knowledge of the RavenClaw dialog management framework, the third system, Vera, was developed as part of a class project, by a team of six students who had no prior experience with RavenClaw. Modulo an initial lack of documentation, no major problems were encountered in configuring the system for automatic error handling. Overall, the proposed error handling architecture adapted easily and provided the desired functionality in each of these domains: while new strategies and recovery policies were developed for some of the systems, no structural changes were required in the error handling architecture.

	RoomLine	Let's Go Public	Vera In / Out
Domain	room reservations	bus route information	task-able agent
Initiative type	mixed	system	mixed / mixed
Task size: #agents ; #concepts	110 ; 25	57 ; 19	29 ; 4 / 31 ; 13
Strategies for misunderstandings	explicit and implicit	explicit	explicit and implicit / explicit only
Policy for misunderstandings	optimistic	always-explicit	optimistic / always-explicit
Strategies for non-understandings	all strategies (see Section 3.1)	go-to-quieter-place, you-can-say, help	all strategies / repeat prompt
Policy for non-understandings	choose-random	author-specified heuristic policy	choose-random / always-repeat-prompt
Sessions collected so far	1393	2836	72 / 131
Avg. task success rate	75%	52%	(unknown)
% Misunderstandings	17%	28%	(unknown)
% Non-understandings	13%	27%	(unknown)
% turns when strategies engage	41%	53%	36% / 44%

Table 1: Spoken dialog systems using the RavenClaw error handling architecture

5 Conclusion and Future Work

We have described the error handling architecture underlying the RavenClaw dialog management framework. Its design is modular: the error handling strategies as well as the mechanisms for engaging them are decoupled from the actual dialog task specification. This significantly lessens the development effort: system authors focus exclusively on the domain-specific dialog control logic, and the error handling behaviors are generated transparently by the error handling process running in the core dialog engine. Furthermore, we have argued that the distributed nature of the error handling process leads to good scalability properties and facilitates the reuse of policies within and across systems and domains.

The proposed architecture represents only the first (but an essential step) in our larger research program in error handling. Together with the systems described above, it sets the stage for a number of current and future planned investigations in error detection and recovery. For instance, we have recently conducted an extensive investigation of non-understanding errors and the ten recovery strategies currently available in the RavenClaw framework. The results of that study fall beyond the scope of this paper and are presented separately in (Bohus and Rudnicky, 2005a). In another project supported by this architecture, we have developed a model for updating system beliefs over concept values in light of initial recognition confidence scores and subsequent user responses to system actions. Initially, our confirmation strategies used simple heuristics to update the system's confidence score for a concept in light of the user response to the verification question. We have showed that a machine learning based approach which integrates confidence information with correction detection information can be used to construct significantly more accurate system beliefs (Bohus and Rudnicky, 2005b). Our next efforts will focus on using reinforcement learning to automatically derive the error recovery policies.

References

Bohus, D., Rudnicky, A., 2003 – *RavenClaw: Dialogue Management Using Hierarchical Task Decomposition and an Expectation Agenda*, in Proceedings of Eurospeech-2003, Geneva, Switzerland

Bohus, D., Rudnicky, A., 2005a – *Sorry, I didn't Catch That! An Investigation into Non-understandings and Recovery Strategies,* to appear in SIGDial-2005, Lisbon, Portugal

Bohus, D., Rudnicky, A., 2005b – *Constructing Accurate Beliefs in Spoken Dialog Systems,* submitted to ASRU-2005, Cancun, Mexico

Clark, H.H., Schaefer, E.F., 1989 – *Contributing to Discourse*, in Cognitive Science, vol 13, 1989.

Kawahara, T., Komatani, K., 2000 – *Flexible mixed-initiative dialogue management using concept-level confidence measures of speech recognizer output*, in Proc. of COLING, Saarbrucken, Germany, 2000.

Krahmer, E., Swerts, M., Theune, M., Weegels, M., 1999 - *Error Detection in Human-Machine Interaction*, Speaking. From Intention to Articulation, MIT Press, Cambridge, Massachusetts, 1999

Paek, T., Horvitz, E., 2000 – *Conversation as Action Under Uncertainty*, in Proceedings of the Sixteenth Conference on Uncertainty and Artificial Intelligence, Stanford, CA, June 2000.

Raux, A., Langner, B., Bohus, D., Black, A., Eskenazi, M., 2005 – *Let's Go Public! Taking a Spoken Dialog System to the Real World*, submitted to Interspeech-2005, Lisbon, Portugal

RoomLine web site, as of June 2005 – www.cs.cmu.edu/~dbohus/RoomLine

Sanders, G., Le, A., Garofolo, J., 2002 – *Effects of Word Error Rate in the DARPA Communicator Data During 2000 and 2001*, in Proceedings of ICSLP'02, Denver, Colorado, 2002.

Scheffler, K., Young, S., 2002 – *Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning*, in Proceedings of HLT-2002.

Singh, S., Litman, D., Kearns, M., Walker, M., 2000 – *Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System*, in Journal of Artificial Intelligence Research, vol. 16, pp 105-133, 2000.

Traum, D., 1998 – *On Clark and Schaefer's Contribution Model and its Applicability to Human-Computer Collaboration*, in Proceedings of the COOP'98, May 1998.

Walker, M., Wright, J., Langkilde, I., 2000 – *Using Natural Language Processing and Discourse Features to Identify Understanding Errors in a Spoken Dialogue System*, in Proc. of the 17'th International Conference of Machine Learning, pp 1111-1118.