Available online at www.sciencedirect.com
ScienceDirect
Computer Speech and Language 23 (2009) 332–361
COMPUTER SPEECH AND LANGUAGE
www.elsevier.com/locate/csl

The RavenClaw dialog management framework: Architecture and systems

Dan Bohus [a,*,1], Alexander I. Rudnicky [b]

[a] *Adaptive Systems and Interaction, Microsoft Research, One Microsoft Way, Redmond, WA 98052, United States*
[b] *Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, United States*

Received 26 July 2008; received in revised form 21 October 2008; accepted 23 October 2008
Available online 5 November 2008

Abstract

In this paper, we describe RavenClaw, a plan-based, task-independent dialog management framework. RavenClaw isolates the domain-specific aspects of the dialog control logic from domain-independent conversational skills, and in the process facilitates rapid development of mixed-initiative systems operating in complex, task-oriented domains. System developers can focus exclusively on describing the dialog task control logic, while a large number of domain-independent conversational skills such as error handling, timing and turn-taking are transparently supported and enforced by the RavenClaw dialog engine. To date, RavenClaw has been used to construct and deploy a large number of systems, spanning different domains and interaction styles, such as information access, guidance through procedures, command-and-control, medical diagnosis, etc. The framework has easily adapted to all of these domains, indicating a high degree of versatility and scalability.

Keywords: Dialog management; Spoken dialog systems; Error handling; Focus shifting; Mixed-initiative

1. Introduction

Over the recent years, advances in automatic speech recognition, as well as language understanding, generation, and speech synthesis have paved the way for the emergence of complex, task-oriented conversational spoken language interfaces. Examples include: Jupiter (Zue et al., 2000) provides up-to-date weather information over the telephone; CMU Communicator (Rudnicky et al., 1999) acts as a travel planning agent and can arrange multi-leg itineraries and make hotel and car reservations; TOOT (Swerts et al., 2000) gives spoken access to train schedules; Presenter (Paek and Horvitz, 2000) provides a continuous listening command and control interface to PowerPoint presentations; WITAS (Lemon et al., 2002) provides a spoken language interface to an autonomous robotic helicopter; AdApt (Gustafson et al., 2000) provides real-estate information in

* Corresponding author.
E-mail address: dbohus@microsoft.com (D. Bohus).
[1] Work conducted while at Carnegie Mellon University.

0885-2308/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.csl.2008.10.001

the Stockholm area; TRIPS (Ferguson and Allen, 1998) is a spoken-language enabled planning assistant. These are only a few; for a longer list, see (Bohus, 2007).

Architecturally, these systems are designed around several key components typically connected in a pipeline architecture, such as the one shown in Fig. 1. The audio signal from the user is captured and passed through a speech recognition module that produces a recognition hypothesis. This recognition hypothesis is then forwarded to a language understanding component that creates a corresponding semantic representation. This semantic input is then passed to the dialog manager, which, based on the current input and discourse context, produces the next system action (typically in the form of a semantic output). A language generation module then produces the corresponding surface (textual) form, which is subsequently passed to a speech synthesis module and rendered as audio back to the user.

The dialog manager therefore plays a key control role in any conversational spoken language interface: given the decoded semantic input corresponding to the current user utterance and the current discourse context, it determines the next system action. In essence, the dialog manager is responsible for planning and maintaining the coherence, over time, of the conversation. Several tasks must be performed in order to accomplish this goal successfully.

First, the dialog manager must maintain a history of the discourse and use it to interpret the perceived semantic inputs in the current context. Second, a representation – either explicit or implicit – of the system task is typically required. The current semantic input, together with the current dialog state and information about the task to be performed is then used to determine the next system action. As we shall see later, different theories and formalisms have been proposed for making these decisions. In some dialog managers, a predefined universal plan for the interaction exists, i.e., the system actions are predetermined for any given user input. Other systems make certain assumptions about the structure of the interaction and dynamically plan the next move at run-time, based on generic dialog rules. Often, dialog managers have to also interact with various domain and application-specific agents. For instance, a dialog system that assists users in making flight reservations must communicate with a database to obtain information and to perform the required transactions.

In guiding the conversation, the dialog manager must also be aware of, and must implement a number of generic conversational mechanisms. A first example is timing and turn-taking. Most systems make the assumption that the two participants in the conversation make successive contributions to the dialog. More complex models need to be developed to support barge-ins, backchannels, or multi-participant conversation.

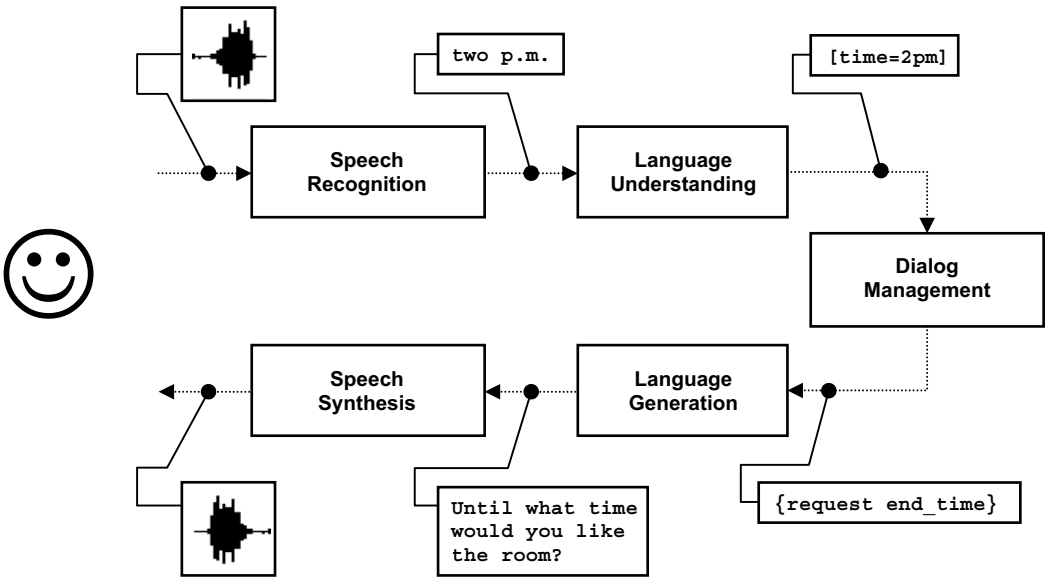

Fig. 1. A classical dialog system architecture.

Another example is error handling. In speech-based conversational systems, the dialog management component must be able to take into account the underlying uncertainties in the recognition results and plan the conversation accordingly. Unless robust mechanisms for detecting and recovering from errors are present, speech recognition errors can lead to complete breakdowns in interaction. Other generic conversational skills include the ability to respond appropriately to various requests like "can you repeat that?", "wait a second", etc.

In this paper, we describe RavenClaw, a plan-based, task-independent dialog management framework. One of the key features of this framework is that it isolates the domain-specific aspects of the dialog control logic from domain-independent conversational skills, and in the process it facilitates the rapid development of mixed-initiative systems operating in complex, task-oriented domains. System developers can focus exclusively on describing the dialog task control logic, while a large number of domain-independent conversational skills such as error handling, timing and turn-taking are transparently supported and enforced by the RavenClaw dialog engine.

The paper is organized as follows: we begin with a brief review of current dialog management technologies in Section 3, and we outline the main objectives that have guided the development of RavenClaw in Section 3. Next, in Section 4, we describe the overall architecture of the RavenClaw dialog management framework, and discuss the various algorithms and data-structures that govern its function and confer the desired properties. In Section 5 we describe a number of systems built using RavenClaw and discuss our experience in developing and deploying this systems. Finally, in Section 6 we present a number of concluding remarks and discuss directions for further extending the framework.

2. Current dialog management technologies

A number of different solutions for the dialog management problem have been developed to date in the community. Some of the most widely used techniques are: finite-state, form-filling, information-state-update, and plan-based approaches. Each of these approaches makes different assumptions about the nature of the interaction; each has its own advantages and disadvantages. In this section, we briefly introduce each of these technologies, to provide the background for the rest of the presentation. A more in-depth review of these technologies falls outside the scope of this paper; for the interested reader, McTear (2002) provides a more comprehensive survey.

In a finite-state dialog manager, the flow of the interaction is described via a finite-state automaton. At each point in the dialog, the system is in a certain state (each state typically corresponds to a system prompt). In each state, the system expects a number of possible responses from the user; based on the received response, the system transitions to a new state. To develop a dialog manager for a new application, the system author must construct the corresponding finite state automaton. In theory, the finite-state automaton representation is flexible enough to capture any type of interaction. In practice, this approach is best suited only for implementing relatively simple systems that retain the initiative throughout the conversation. In these cases, the finite-state automaton representation is very easy to develop, interpret, and maintain. However, the finite-state representation does not scale well for more complex applications or interactions. For instance, in a mixed-initiative system (where the user is also allowed to direct and shift the focus of the conversation), the number of transitions in the finite-state automaton grows very large; the representation becomes difficult to handle. One representative example of this approach is the CSLU dialog management toolkit (Cole, 1999; Understanding, 2007).

Another dialog management technology, especially useful in information access domains is form-filling (also known as slot-filling). In this case the basis for representing the system's interaction task is the form (or frame). A form consists of a collection of slots, or pieces of information to be collected from the user. For instance, in a train schedule information system, the slots might be the departure and arrival city, the travel date and time. Each slot has an associated system prompt that will be used to request the corresponding information from the user. Typically, an action is associated with each form, for instance access to the schedule database in the train system. The system guides the dialog such as to collect the required slots from the user (some of the slots might be optional); the user can also take the initiative and provide information about slots that the system has not yet asked about. Once all the desired information is provided, the system performs the action associated with the form. The system's interaction task may be represented as a collection

of chained forms, with a specified logic for transitioning between these forms. In comparison with the finite-state representation, the form-filling approach makes stronger assumptions about the nature of the interaction task, and in the process allows system authors to more easily specify it. As we have mentioned before, this approach is well-suited in information access domains where the user provides some information to the system, and the system accesses a database or performs an action based on this information. However, the approach cannot be easily used to construct systems in domains with different interaction styles: tutoring, guidance, message delivery, etc. Representative examples of this approach are the industry standard VoiceXML (VoiceXML, 2007), and the Phillips' SpeechMania system (Aust and Schroer, 1998).

A third dialog management approach that has recently received a lot of attention and wide adoption in the research community is information-state-update (ISU). In this approach the interaction flow is modeled by a set of update rules that fire based on the perceived user input and modify the system's state. The system state (also known as information-state) is a data structure that contains information accumulated throughout the discourse. The information-state-update approach allows for a high of flexibility in managing the interaction. Different ISU systems can capture different information in the state, and implement different linguistic theories of discourse in the state-update rules. A potential drawback of this approach is that, as the set of update rules increases, interactions between these rules and their overall effects become more difficult to anticipate. Representative examples of the ISU approach include the TrindiKit dialog move engine (Larsson, 2002; TrindiKit, 2007) and DIPPER (Bos et al., 2003).

The fourth dialog management technology we have mentioned are plan-based approaches. In this case, the system models the goals of the conversation, and uses a planner to guide the dialog along a path towards these goals. These systems reason about user intentions, and model relationships between goals and sub-goals in the conversation, and the conversational means for achieving these goals. As a consequence, they require more expertise from the system developer, but can enable the development of more complex interactive systems. Examples include the TRAINS and TRIPS systems (Ferguson and Allen, 1998), and Collagen (Rich and Sidner, 1998; Rich et al., 2001). The RavenClaw dialog manager we describe in the rest of this paper bares most similarities to this last class of systems, following essentially a hierarchical plan-based approach.

3. Objectives

The RavenClaw dialog management framework was developed as a successor of the earlier Agenda dialog management architecture used in the CMU Communicator project (Rudnicky et al., 1999). The primary objective was to develop a robust platform for research in dialog management and conversational spoken language interfaces. In support of this goal, we identified and pursued several desirable characteristics:

3.1. Task-independence

The dialog management framework should provide a clear separation between the domain-specific aspects of the dialog control logic and domain-independent, reusable dialog control mechanisms. This decoupling will significantly lessen the system development effort and promotes reusability of various solutions and components. System authors focus exclusively on describing the domain-specific aspects of the dialog control logic, while a reusable, domain-independent dialog engine transparently supports and enforces a number of generic conversational skills (e.g., error handling, timing and turn-taking, context establishment, help, etc.).

3.2. Flexibility

The framework should accommodate a wide range of application domains and interaction styles. Some dialog management formalisms are more suited for certain types of applications. For instance, form-filling approaches are typically well-suited in information access domains; however, the form-filling paradigm does not easily support the development of a tutoring application. The RavenClaw dialog management framework uses a hierarchical plan-based formalism to represent the dialog task. We have found that this representation provides a high degree of flexibility, while also providing good scalability properties. To date, RavenClaw has been used to construct over a dozen dialog systems spanning different domains and interaction styles: infor-

mation-access, guidance through procedural tasks, message-delivery, command-and-control, web-search, scheduling. More details about the various systems will be presented later, in Section 5.

3.3. Transparency

The framework should provide access to detailed information about each of its internal subcomponents, states and run-time decisions. RavenClaw supports configurable multiple-stream logging: each of its subcomponents generates a rich log stream containing information about internal state, computations, and decisions made. Furthermore, a number of task-independent data analysis and visualization tools have been developed.

3.4. Modularity and reusability

Specific functions, for instance dialog planning, input processing, output processing, error-handling, etc. should be encapsulated in subcomponents with well-defined interfaces that are decoupled from domain-specific dialog control logic. RavenClaw users are able to inspect and modify each of these components individually, towards their own ends. Modularity promotes the reusability and portability of the developed solutions. For instance, error handling strategies developed in the context of a system that helps users make conference room reservations (Bohus and Rudnicky, 2003) were later introduced into a system that provides bus schedule information (Raux et al., 2005). The RavenClaw dialog management framework was adapted to work with different types of semantic inputs, e.g., generated by the Phoenix parser (Ward and Issar, 1994) or the Gemini parser (Dowding et al., 1993), by simply overwriting the input processing class.

3.5. Scalability

The framework should support the development of practical, real-world spoken language interfaces. While simple, well-established approaches such as finite-state call-flows allow the development of practical, large-scale systems in information access domains, this is usually not the case for frameworks that provide the flexibility and transparency needed for research. At the same time, a number of relevant research questions do not become apparent until one moves from toy systems into large-scale applications. In RavenClaw, the hierarchical plan-based representation of the domain-specific dialog control logic confers good scalability properties, while at the same time it does not sacrifice flexibility or transparency.

3.6. Open-source

Together with a number of end-to-end spoken dialog systems developed within this framework, complete source code for RavenClaw has been released under an open-source license (RavenClaw-Olympus, 2007). We hope this will foster further research and contributions from other members of the research community.

4. The RavenClaw dialog management framework

RavenClaw is a two-tier dialog management framework that enforces a clear separation between the domain-dependent and the domain-independent aspects of the dialog control logic – see Fig. 2. The domain-specific aspects are captured by the dialog task specification, essentially a hierarchical-plan for the interaction, provided by the system author. A reusable, domain-independent dialog engine manages the conversation by executing the given dialog task specification. In the process, the dialog engine also contributes a basic set of domain-independent conversational strategies such as error handling, timing and turn-taking behaviors, and a variety of other universal dialog mechanisms, such as help, repeat, cancel, suspend/resume, quit, start-over, etc.

The decoupling between the domain-specific and domain-independent aspects of dialog control provides a number of benefits: it significantly lessens the system authoring effort, it promotes reusability and portability, and it ensures consistency and uniformity in behavior both within and across domains. System developers can focus exclusively on describing the dialog control logic, while a large number of domain-independent

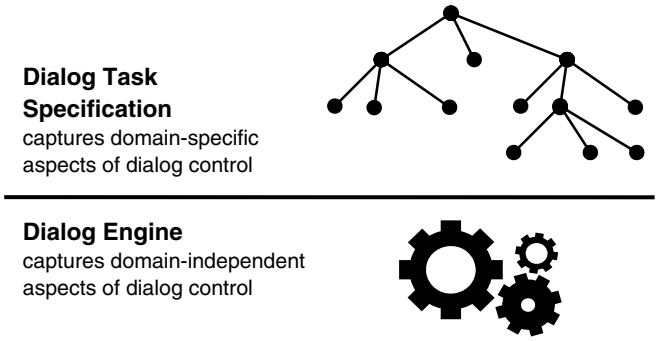

Fig. 2. RavenClaw – a two-tier dialog management architecture.

conversational skills are transparently supported and enforced by the dialog engine. Consider for instance error handling. System developers construct a dialog task specification under the assumption that inputs to the system will always be perfect, therefore ignoring the underlying uncertainties in the speech recognition channel. The responsibility for ensuring that the system maintains accurate information and that the dialog advances normally towards its goals is delegated to the dialog engine.

In the rest of this section, we describe in more depth the RavenClaw dialog management architecture. We begin by discussing the dialog task specification language in the next subsection. Then, in Section 4.2, we describe the algorithms and data-structures which govern the dialog engine. Next, in Section 4.3 we describe the error handling mechanisms implemented as part of the engine. Finally, in Section 4.4, we present a number of additional task-independent conversational strategies that are automatically supported in the RavenClaw framework.

4.1. The dialog task specification

The dialog task specification in the RavenClaw dialog management framework captures the domain-specific aspects of the dialog control logic. Developing a new dialog manager using the RavenClaw framework therefore amounts to writing a new dialog task specification.

The dialog task specification describes a hierarchical plan for the interaction. More specifically, a dialog task specification consists of a tree of dialog agents, where each agent is responsible for handling a subpart of the interaction. For instance, Fig. 3 depicts the top portion of the dialog task specification for RoomLine, a spoken dialog system which can assist users in making conference room reservations (more details about this system will be presented later, in Section 5.2). The root node subsumes several children: `Login`, which identifies the user to the system, `GetQuery`, which obtains the time and room constraints from the user, `GetResults`, which executes the query against the backend, and `DiscussResults` which presents the obtained

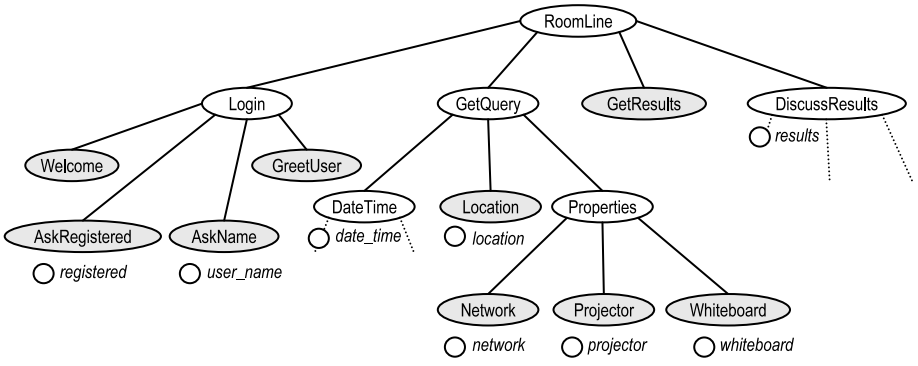

Fig. 3. A portion of the dialog task tree for the RoomLine system.

results and handles the forthcoming negotiation for selecting the conference room that best matches the user's needs. Moving one level deeper in the tree, the `Login` agent decomposes into `Welcome`, which provides a short welcome prompt, `AskRegistered` and `AskName`, which identify the user to the system, and finally `GreetUser`, which sends a greeting to the user.

The dialog agents in a dialog task specification fall into two categories: **fundamental dialog agents**, shown grayed in Fig. 3, and **dialog-agencies**, shown in clear in Fig. 3. The **fundamental dialog agents** are located at the terminal positions in the tree (e.g. `Welcome`, `AskRegistered`) and implement atomic dialog actions, or dialog moves. There are four types of fundamental dialog agents: **Inform** – produces an output (e.g., `Welcome`), **Request** – requests information from the user (e.g., `AskRegistered`), **Expect** – expects information from the user, but without explicitly requesting it (e.g., `Projector`) and **Execute**- performs a domain-specific operation, such as database access (e.g. `GetResults`). The **dialog-agencies** occupy non-terminal positions in the tree (e.g., `Login`, `GetQuery`); their purpose is to control the execution of their subsumed agents, and encapsulate the higher level temporal and logical structure of the dialog task.

Each dialog agent implements an `Execute` routine, which is invoked at runtime by the dialog engine. The `Execute` routine is specific to the agent type. For example, inform-agents generate an output when executed, while request-agents generate a request but also collect the user's response. For dialog-agencies, the Execute routine is in charge of planning the execution of their subagents. Besides the `Execute` routine, each dialog agent can define **preconditions**, **triggers**, as well as **success** and **failure criteria**. These are taken into account by the dialog engine and parent dialog-agencies while planning the execution of the various agents in the tree.

If the dialog agents are the fundamental execution units in the RavenClaw dialog management framework, the data that the system manipulates throughout the conversation is encapsulated in **concepts**. Concepts can be associated with various agents in the dialog task tree, for instance `registered` and `user_name` in Fig. 3, and can be accessed and manipulated by any agent in the tree. Several basic concept types are predefined in the RavenClaw dialog management framework: Boolean, string, integer and float. Additionally, the framework provides support for more complex, developer–defined concept types such as (nested) structures and arrays. Internally, the "value" for each concept is represented by a set of value/confidence pairs, for instance `city_name = {Boston/0.35; Austin/0.27}`. The dialog engine can therefore track multiple alternate hypotheses for each concept, and can capture the level of uncertainty in each hypothesis (Bohus and Rudnicky, 2005a; Bohus and Rudnicky, 2006). Additionally, each concept also maintains the history of previous values, as well as information about the grounding state, when the concept was last updated, etc.

In practice, the dialog task specification is described by the system author using an extension of the C++ language constructed around a set of predefined macros. Fig. 4 illustrates a portion of the dialog task specification for the RoomLine system, corresponding to the `Login` sub-tree. Each agent in the dialog task tree is specified using a define agent directive (e.g. `DEFINE_AGENCY`, `DEFINE_REQUEST_AGENT`, etc.). For instance, the lines from 1 to 10 define the `Login` dialog-agency. This agency stores a Boolean concept which indicates whether the user is registered with the system or not – `registered`, and a string concept that will contain the user's name – `user_name`. The agency has four subagents: `Welcome`, `AskRegistered`, `AskName` and `GreetUser`, and succeeds when the `GreetUser` agent has completed. Lines 13–15 define the `Welcome` inform-agent. This agent sends a non-interruptible output prompt that will welcome the user to the system. Next, lines 17–20 define the `AskRegistered` request-agent. When executed, this agent asks whether the user is registered with the system or not. It expects a `[Yes]` or a `[No]` semantic answer, and, upon receiving such an answer, it will fill in the registered concept with the appropriate value – true or false. Similarly, the `AskName` agent defined in lines 22–26 asks for the `user_name` concept. Note that this agent also has a precondition, i.e. that the registered concept is true. If the user answered that she is not registered with the system this agent will be skipped during execution. Finally, lines 28–31 define the `GreetUser` inform-agent. This agent sends out a greeting prompt; the values of the `registered` and `user_name` concepts are also sent as parameters to the language generation module.

The various macros shown in this example (e.g., `DEFINE_AGENCY`, `DEFINE_REQUEST_AGENT`, `REQUEST_CONCEPT`, `GRAMMAR_MAPPING`, etc.) are expanded at compile-time by the C++ preprocessor and a class is generated for each defined dialog agent. The directives and parameters specified for each agent (e.g. `PROMPT`, `REQUEST_CONCEPT`, `GRAMMAR_MAPPING`, etc.) are expanded into methods that overwrite virtual methods from the base class, in effect customizing the agent to implement the desired behavior.

```
DEFINE_AGENCY( CLogin,
  DEFINE_CONCEPTS(
    BOOL_USER_CONCEPT( registered, "default" )
    STRING_USER_CONCEPT( user_name, "default" )
  DEFINE_SUBAGENTS(
    SUBAGENT( Welcome, CWelcome, "" )
    SUBAGENT( AskRegistered, CAskRegistered, "default" )
    SUBAGENT( AskName, CAskName, "default" )
    SUBAGENT( GreetUser, CGreetUser, "" ))
  SUCCEEDS_WHEN(COMPLETED(GreetUser))
)

DEFINE_INFORM_AGENT( CWelcome,
  PROMPT(":non-interruptible inform welcome")
)

DEFINE_REQUEST_AGENT( CAskRegistered,
  REQUEST_CONCEPT(registered)
  GRAMMAR_MAPPING("[Yes]>true, [No]>false")
)

DEFINE_REQUEST_AGENT( CAskName,
  PRECONDITION(IS_TRUE(registered))
  REQUEST_CONCEPT(user_name)
  GRAMMAR_MAPPING("[UserName]")
)

DEFINE_INFORM_AGENT( CGreetUser,
  PROMPT("inform greet_user <registered <user_name")
)
```

Fig. 4. A portion of the dialog task specification for the RoomLine system.

To summarize, the dialog task tree specification describes an overall hierarchical plan for the interaction. However, this developer-specified plan does not prescribe a fixed order for the execution of the various dialog agents (as might be found in a directed dialog system). When the dialog engine executes a given dialog task specification, a particular trace through this hierarchical plan is followed, based on the user inputs, the encoded domain constraints and task logic, as well as the various execution policies in the dialog engine. This type of hierarchical task representation has been used for task execution in the robotics community. More recently, this formalism has gained popularity in the dialog management community. Other examples besides RavenClaw include the use of a tree-of-handlers in Agenda Communicator (Rudnicky et al., 1999), activity trees in WITAS (Lemon et al., 2002) and recipes in Collagen (Rich and Sidner, 1998; Rich et al., 2001). In the context of spoken dialog systems, hierarchical plan-based representations present several advantages. Most goal-oriented dialog tasks have an identifiable structure which naturally lends itself to a hierarchical description. The subcomponents are typically independent, leading to ease in design and maintenance, as well as good scalability properties. The tree representation captures the nested structure of dialog and thus implicitly represents context (via the parent relationship), as well as a default chronological ordering of the actions (i.e., left-to-right traversal). Finally, the tree structure can be extended at runtime, and allows for the dynamic construction of dialog structure, a very useful feature for certain types of tasks.

4.2. The RavenClaw dialog engine

We now turn our attention to the algorithms used by the RavenClaw dialog engine to execute a given dialog task specification. The dialog engine algorithms are centered on two data-structures: a **dialog stack**, which captures the discourse structure at runtime, and an **expectation agenda**, which captures what the system expects to hear from the user in any given turn. The dialog is controlled by interleaving **Execution Phases** with **Input Phases** – see Fig. 5. During the execution phase, dialog agents from the task tree are placed on and executed from the dialog stack, generating in the process the system behavior. During the input phase, the system uses

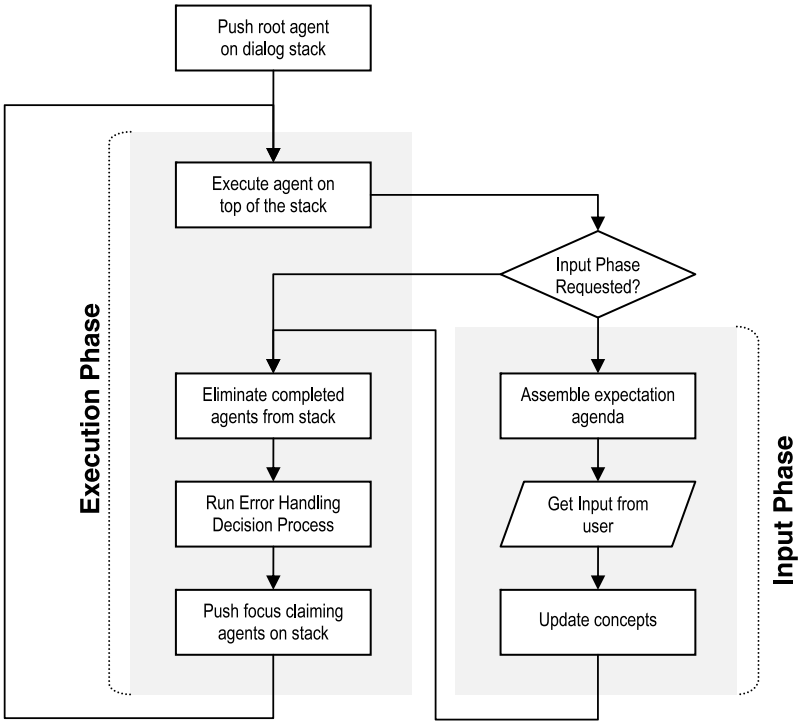

Fig. 5. Block diagram for core dialog engine routine.

the expectation agenda to transfer information from the current user input into the concepts defined in the dialog task tree. Below, we describe in more detail each of these two phases.

4.2.1. The execution phase

During the execution phase the RavenClaw dialog engine performs a number of operations in succession – see Fig. 5.

First, the dialog engine invokes the `Execute` routine of the agent on top of the dialog stack. The effects of the `Execute` routine are different from one agent-type to another. For instance, inform-agents output a system prompt; request-agents output a system request and then request an input phase; dialog-agencies push one of their subagents on the dialog stack. Once the `Execute` routine completes, the control is returned to the dialog engine. If no input phase was requested (some agents can make this request upon completing the `Execute` routine), the dialog engine tests the completion conditions for all the agents on the dialog stack. Any completed agents are eliminated from the dialog stack. Next, the dialog engine invokes the error handling decision process. In this step, the error handling decision process (which we will describe in more detail in Section 4.3) collects evidence about how well the dialog is proceeding, and decides whether or not to trigger an error handling action. If an error recovery action is necessary, the error handling decision process dynamically creates and pushes an error handling agency (e.g., explicit confirmation, etc.) on the dialog stack. Finally, in the last stage of the execution phase, the dialog engine inspects the focus claims (trigger) conditions for all the agents in the dialog task tree. If any agents in the task tree request focus, they will be pushed on top of the dialog stack.

For clarity, we will present a step-by-step trace through the execution of the RoomLine dialog task in Fig. 6. The corresponding dialog task tree is also shown in the same figure. At start-up, the dialog engine places the root agent, `RoomLine` in this case, on the dialog stack. Next, the dialog engine enters an execution phase. First, the engine invokes the `Execute` routine for the agent on top of the stack – `RoomLine`. This is a dialog-agency, which, based on its execution policy and on the preconditions of its subagents, decides that it needs to first engage the `Login` agent. It therefore pushes `Login` on the dialog stack – see Fig. 6, step 2, and returns the control to the dialog engine. Next the dialog engine pops all completed agents from the dialog stack. Since neither

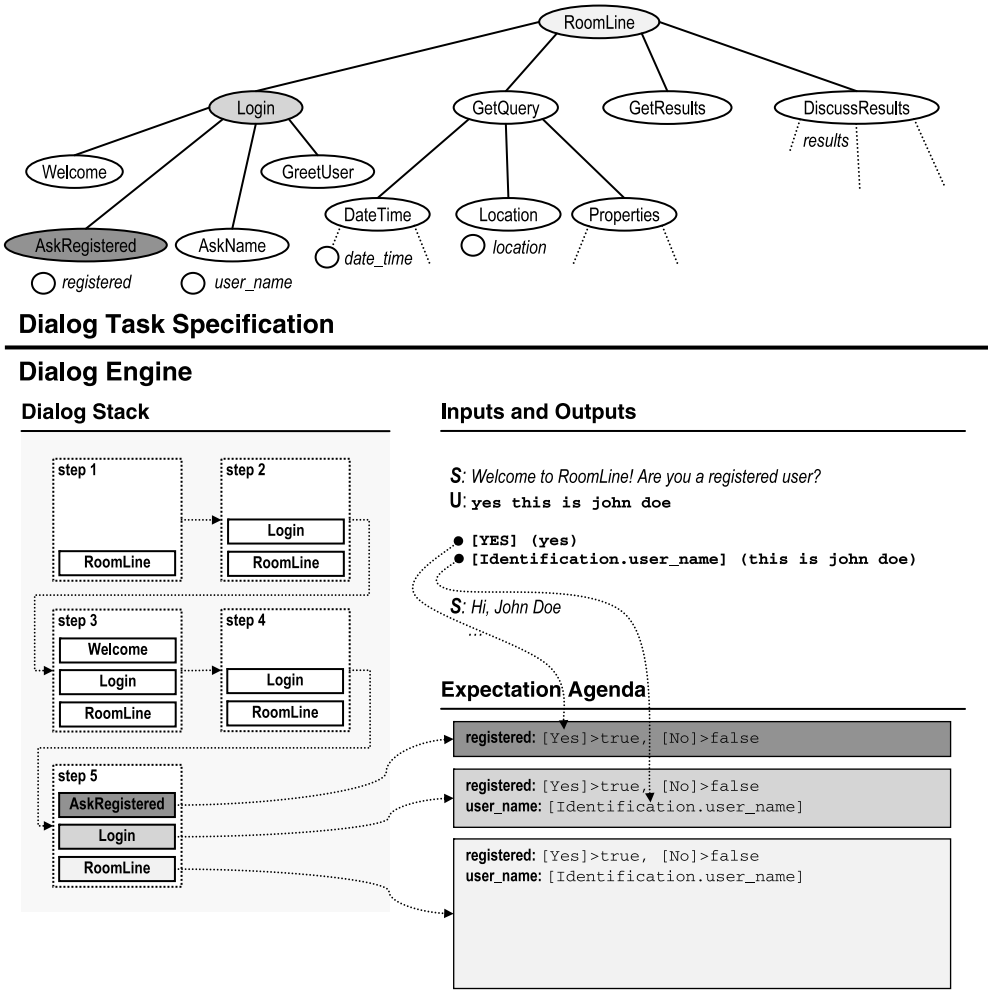

Fig. 6. Execution trace through the RoomLine task.

`RoomLine` nor `Login` is yet completed, the dialog engine continues by invoking the error handling decision process. No error handling actions are taken in this case.[2] Next the dialog engine inspects the focus claims, but no focus claims are present at this point. The dialog engine therefore engages in a new execution phase. This time, `Login` is on top of the stack, so the dialog engine invokes `Login.Execute`. `Login` pushes the `Welcome` agent on the dialog stack and returns the control to the dialog engine – see Fig. 6, step 3. Again no agents are completed, no grounding actions are taken and no focus claims are present. Next, the dialog engine executes `Welcome`. This is an inform-agent, which will send out a welcome message to the user. The system says: "Welcome to RoomLine, the conference room reservation assistant." Next, when the dialog engine inspects the completion conditions, it will find that Welcome has completed (inform-agents complete as soon as they output the prompt), and it will therefore pop `Welcome` from the execution stack – see Fig. 6, step 4. In the next execution phase, `Login.Execute` is invoked again. This time, since the `Welcome` agent is already completed, the `Login` agency will plan the `AskRegistered` agent for execution by pushing it on the dialog stack – see Fig. 6, step 5. Again, none of the agents on the stack are completed, no grounding actions are taken and no focus claims are made. When the dialog engine next executes `AskRegistered`, this agent will output a request – "Are you a

[2] For clarity purposes, we have kept this example simple. For instance, no focus shifts or error handling strategies have been illustrated. In the next subsection, we will present another example that includes a focus shift. Another more complex execution trace which involves the invocation of various error handling strategies is presented later on, in Section 4.3.1.

registered user?", and then invoke an input phase by passing a certain return-code to the dialog engine. The input phase is discussed in the next subsection.

The dialog stack therefore captures the nested structure of the discourse. Note that the isomorphism between the dialog stack and the task tree is only apparent. There is an essential functional difference between the two structures: the dialog stack captures the temporal and hierarchical structure of the discourse, while the tree describes the hierarchical goal structure of the task. Although in the example discussed above the two structures match, this is not always the case. For instance, if the trigger condition for an agent becomes true, the dialog engine will push that agent on top of the dialog stack. The dialog focus will be shifted to that agent. The execution will therefore continue with that agent, and the isomorphism between the stack and the tree will be broken. Once the agent completes and is popped from the stack, we're back to where we were before the focus shift; a concrete example is illustrated in the next subsection.

In general, the agent on top of the stack represents the current focus of the conversation, and, the agents below it (which typically sit above it in the tree) capture successively larger discourse contexts. As we have already seen, the dialog stack provides support for maintaining the context during focus shifts and correctly handling sub-dialogs. Additionally, the dialog stack is used to construct the system's agenda of expectations at every turn, as described in the next section.

4.2.2. The input phase

4.2.2.1. Overview. An input phase is invoked each time a request-agent is executed. In the example discussed above, the input phase was triggered by the execution of the `AskRegistered` agent. Each input phase consists of three stages, also illustrated in Fig. 5: (1) assembling the expectation agenda, (2) obtaining the input from the user, and (3) updating the system's concepts based on this input.

In the first stage, the system assembles the **expectation agenda** – a data-structure that describes what the system expects to hear from the user in the current turn. The agenda is organized into multiple levels. Each level corresponds to one of the agents on the dialog stack, and therefore to a certain discourse segment. The dialog engine traverses the stack, from the top element to the bottom, and constructs the corresponding levels in the expectation agenda. In the example described above, immediately after the `AskRegistered` agent triggers an input phase, the stack contains the `AskRegistered`, `Login` and `RoomLine` agents – see Fig. 6, step 5. The dialog engine therefore constructs the first level of the agenda by collecting the expectations from the `AskRegistered` agent. The `AskRegistered` agent expects to hear a value for registered concept, in the form of either a `[Yes]` or a `[No]` grammar slot in the input. The second level in the agenda is constructed by collecting the expectations from the next agent on the stack, i.e., `Login`. When an agency declares its expectations, by default it collects all the expectations of its subagents. In this case, the Login agency expects to hear both the registered concept (from the `AskRegistered` agent), and the `user_name` concept (from the `AskUserName` agent). Finally, the third and in this case last level of the expectation agenda is constructed by collecting all the expectations from the `RoomLine` agent. Apart from the `registered` and `user_name` concepts, this last level contains the expectations from all other agents in the dialog task tree. In effect, the levels in the expectation agenda encapsulate what the system expects to hear starting from the currently focused question and moving in larger and larger discourse segments.

After the expectation agenda has been assembled, the dialog engine **waits for an input** from the user; this is the second stage of the input phase.

Finally, once the input arrives, the dialog engine engages in a **concept binding** stage. In this step, the information available in the input is used to update system concepts. The updates are governed by the expectation agenda. The dialog engine performs a top-down traversal of the agenda, looking for matching grammar slots in the user input. Wherever a match is found, the corresponding concept is updated accordingly. For instance, in the example from Fig. 6, the recognized user response is *"Yes this is John Doe*", which is parsed as

```
[YES] (yes) [UserName] (john doe)
```

In this case, the `[YES]` slot matches the expectation for the registered concept on the first level in the agenda, and the `[UserName]` slot matches the expectation for the `user_name` concept on the second level in the agenda. These two concepts will be updated accordingly: the `registered` concept will contain `true`, and the `user_name` concept will contain `''john doe''`. The concept updating process relies on a set of belief

updating operators (Bohus and Rudnicky, 2005a; Bohus and Rudnicky, 2006), which take into account information about the initial belief about the concept (i.e., the set of alternate hypotheses and their corresponding confidence scores), the user response, as well as the current context such as the last system action, the number of previous detected errors, etc.

Once the input phase completes, the dialog engine continues with another execution phase. The `AskRegistered` agent will be popped from the dialog stack; this agent has completed since the registered concept is now updated. When the `Login` agent will execute again, it will skip over `AskUserName` since the `user_name` concept is already available – the default precondition on request-agents is that the requested concept is not available. The next agent planned by `Login` will therefore be `GreetUser`, and the system responds: "Hello, John Doe ..."

The expectation-agenda driven concept update mechanism provides a number of advantages: (1) it allows for the user to over-answer system questions, (2) in conjunction with the dialog stack, it provides support for mixed-initiative interaction, (3) it automatically performs context-based semantic disambiguation, and (4) it can provide a basis for dynamic state-specific language modeling. The first aspect was already illustrated in the example discussed above: the user not only answered the system question, but he also provided his name. In the next subsections we discuss in more detail the other three aspects listed above.

4.2.2.2. Expectation agenda and mixed-initiative interaction. The expectation agenda facilitates mixed-initiative conversation, since the system can integrate information from the user's response that does not necessarily pertain to the question in focus. In the previous example, we have illustrated a simple case in which the user over-answers a system question. More generally, in combination with the dialog stack, the expectation agenda allows the user to take initiative and shift the focus of the conversation.

We illustrate the focus-shift mechanism with an example from a spoken dialog system that operates in the air travel planning domain. The example is illustrated in see Fig. 7. At turn n, the system question is "Will you be returning from San Francisco?" corresponding to the `/FlightLine/Legl/AskReturn` agent in the dialog task tree. At this point, instead of responding to the system question, the user decides to ask about booking a particular hotel in San Francisco. The decoded input matches the expectation for `[HotelName]`, on the last level in the agenda, and the `hotel_name` concept is updated accordingly. Once this input phase completes, the system continues with an execution phase. During the focus claims analysis, the `/FlightLine/Hotels` agent claims focus, since this agent has a trigger condition that the `hotel_name` concept is updated. As a consequence, the dialog engine places this agent on top of the dialog stack – see the stack at time $n+1$ in Fig. 7. As the dialog engine continues execution, the conversation continues from the `Hotels` dialog agency. Once the hotels sub-dialog completes, the `Hotels` agency is popped from the execution stack and we're back in the previous context, on the `AskReturn` question.

The system author can control the amount of initiative the dialog manager allows the user to take at every point in the dialog by controlling which expectations on the agenda are open and which expectations are closed (expectations that are closed will not bind). By default, the expectations defined by a request- or an expect-agent are open only when the focus of the conversation is under the same main topic as the agent that defines the expectation. For instance, if our Hotels agent was defined as a **main topic** (using the `IS_MAIN_TOPIC` directive), then the `[HotelName]` expectation would be closed at step n in Fig. 7, and the `hotel_name` concept would not be updated. System authors can therefore control which expectations are open and which are closed by defining the main topics in the tree.

A finer-grained level of control can be achieved through a set of **expectation scope operators**, which can be used to alter the default activation behavior of the expectations:

- the ! operator; when this operator is used while defining an expectation (e.g. `![Yes]>true`), the expectation will be open only when the agent that defines the expectation is actually in focus,
- the * operator; when this operator is used, the expectation is always open,
- the `@ (<agent_name>;<agent_name>; ...)` operator; the expectation is open only when the focus of the conversation is under one of agents in the specified list. For instance, if we wanted to allow the `hotel_name` concept to bind only while the conversation is on the first leg of the trip, but not the second leg of the trip, the expectation could be defined as

  ```
  @(/FlightInfo/Legl;/FlightInfo/Hotels)[HotelName]
  ```

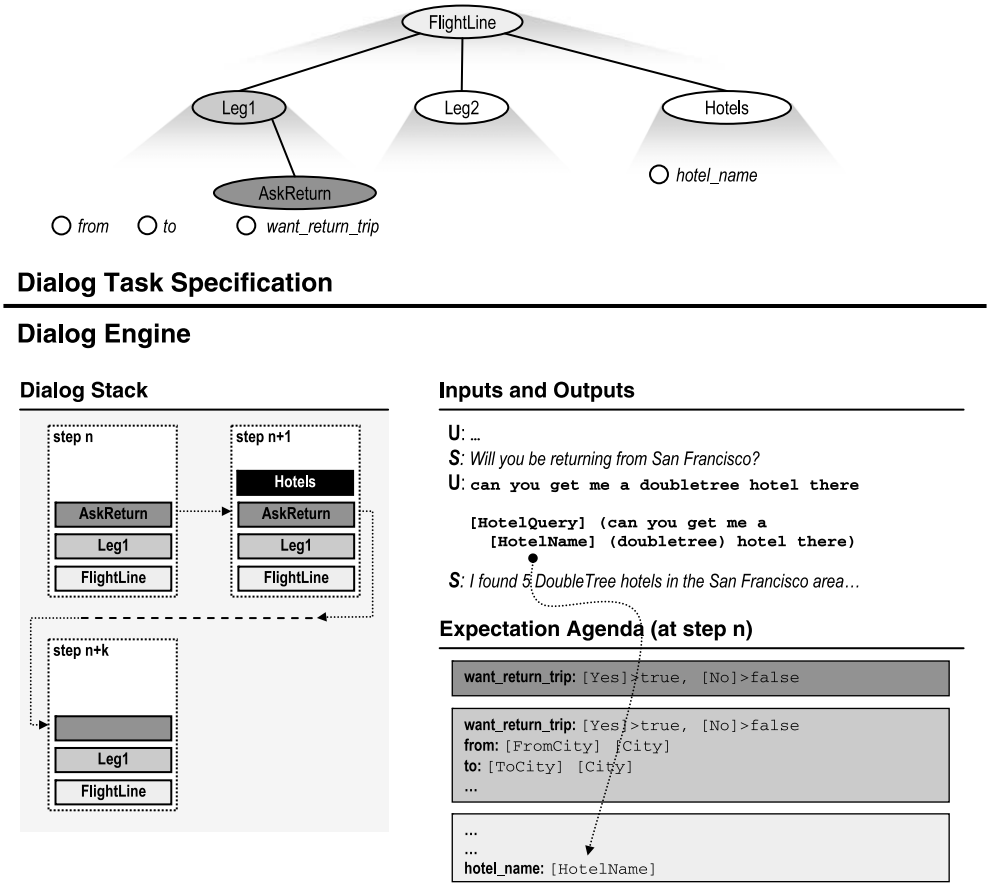

Fig. 7. Focus-shift in a mixed-initiative conversation.

The EXPECT_WHEN macro provides yet another mechanism to control when expectations are open and when they are closed. This macro can be used on request- and expect-agents to define a Boolean condition that describes at which times the expectation should be open. System authors can take into account any (state) information available to the dialog manager in order to control the opening and closing of expectations.

So far, we have discussed how system authors can control the amount of initiative the user is allowed to take at any point in the dialog. Note that the dialog engine could also automatically control the amount of initiative by limiting how deep in the expectation agenda bindings are allowed. For instance, allowing bindings only in the first level in the agenda corresponds to a system-initiative situation where the user is allow to respond only to the question in focus. While this type of behavior has not yet been implemented in the RavenClaw dialog engine, it is easy to envision a system where, perhaps depending on how well the dialog is progressing, the dialog engine automatically adjusts the level of initiative permitted to the user.

4.2.2.3. Expectation agenda and context-based semantic disambiguation. Another advantage provided by the expectation agenda is automatic resolution of some semantic ambiguities based on context. This feature appears as a side-effect of the top-down traversal of the agenda during the concept binding phase.

Consider the example from Fig. 8, again drawn from a fictitious system operating in the air travel domain. The focus of the conversation is on the AskFrom request-agent, which is in charge of obtaining the departure city for the first leg of the trip. This agent declares two expectations for the from_city concept: [FromCity], which captures constructs like for instance "I'd like to leave from San Francisco", and [City] which captures city names spoken in isolation, for instance "San Francisco". At the same time, the AskTo request agent in the Arrival subtree also declares the expectations [ToCity] and [City] in order to capture the arrival city in

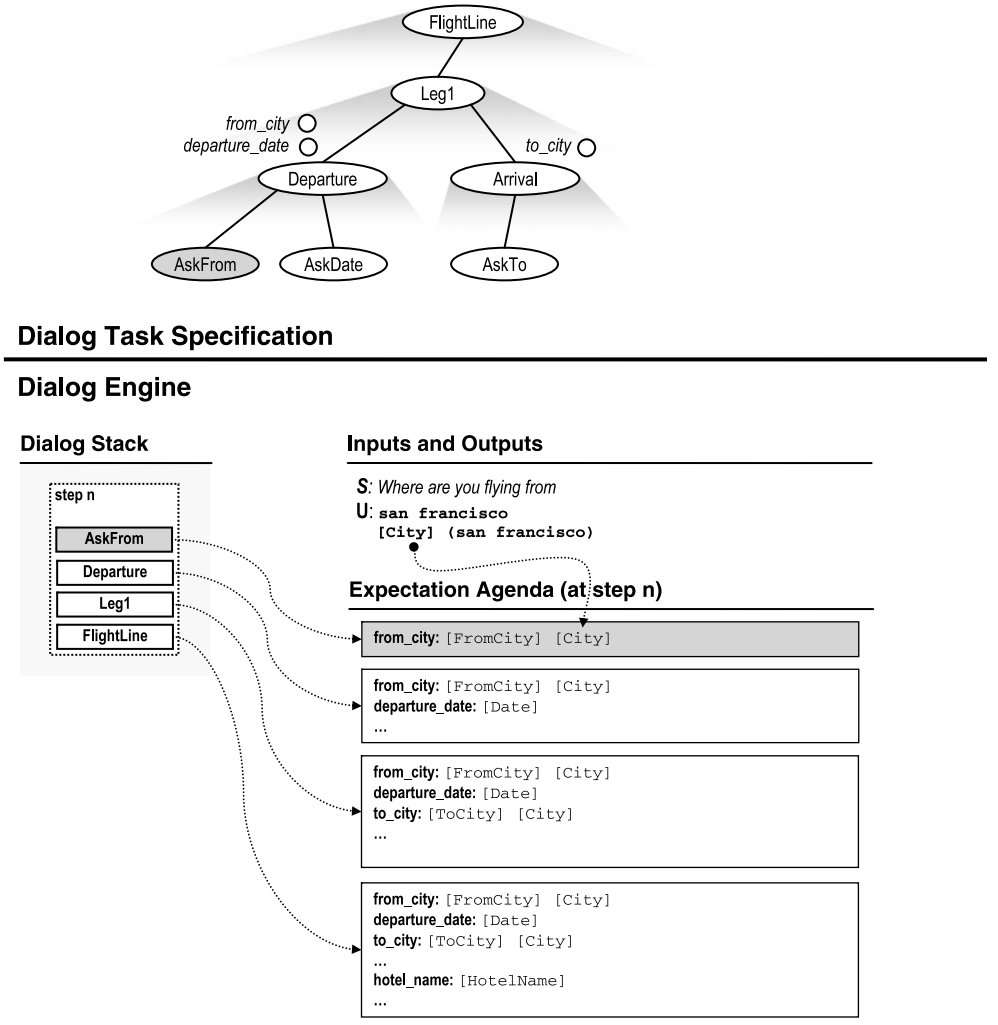

Fig. 8. Context-based semantic disambiguation in the expectation agenda.

the `to_city` concept. The user responds to the system question with a simple city name, which is semantically decoded as `[City]`. A semantic ambiguity arises: should this city bind to the `from_city` concept, or to the `to_city` concept? The ambiguity is automatically resolved given the top-down traversal of the agenda in the concept binding phase. In this case, the input updates the `from_city` concept, since this appears on the higher (in this case first) level in the agenda.

The expectation agenda therefore automatically implements an ambiguity resolution heuristic: if an input could be used to update more than one concept, always update the concept that is closest to the current context, i.e., higher in the agenda, which we believe mimics the heuristic used in human-human conversation.

4.2.2.4. Expectation agenda and dynamic state-specific language modeling. The expectation agenda can also support dynamic, context-specific language modeling. At each turn in the dialog, the expectation agenda captures what the system expects to hear from the user, at the semantic level. This information could be used to dynamically construct a context-specific recognition language-model by interpolating a large number of smaller, fixed language models, and thereby improve recognition accuracy (Xu and Rudnicky, 2000). For instance, considering the example from Fig. 8, the system could create a state-specific language model by interpolating models for `[Yes]`, `[No]`, `[FromCity]`, `[ToCity]`, `[City]`, etc.

The level-based organization of the expectation agenda provides additional information about the likelihood of different user responses. The weights in the interpolation could be assigned based on the depth of

the corresponding item in the expectation agenda. An advantage of this type of interpolated language models is that they can take into account the current context and dialog history in a more accurate fashion than a simple state-specific language model would (the expectation agenda might contain different grammar slots when a certain agent is in focus, depending on the dialog history). Secondly, the approach would not require the models to be retrained after each change to the dialog task structure. While this type of context-sensitive language-modeling approach has not yet been implemented in the RavenClaw dialog engine, (Gruenstein et al., 2005) have shown considerable reductions in word-error-rate under certain circumstances with a similar technique.

4.3. Error handling in the RavenClaw dialog management framework

In the previous section we described the data-structures and algorithms that govern the execution of a dialog task by the RavenClaw dialog engine. The discussion so far has made the implicit assumption that the inputs to the system are perfect, i.e. that they accurately capture the user's expressed intent. This is however not always the case. While significant advances were made in the last decades, current speech recognition and language understanding technologies are still far from perfect. Typical recognition error-rates in all but the most simple systems range from 15% to as high as 40% or even higher for non-native populations or under adverse noise conditions.

Left unchecked, speech recognition errors can lead to two types of understanding-errors: **non-understandings** and **misunderstandings**. A **non-understanding** is said to occur when the system fails to acquire any useful information from the user's turn. For instance, if the parsing component lacks robustness, then even the smallest recognition error can ruin the whole language understanding process. In such cases no meaningful semantic representation of the input can be obtained, and we say we have a non-understanding. In contrast, a **misunderstanding** occurs when the system extracts some information from the user's turn, but that information is incorrect. This generally happens when the language understanding layer generates a plausible semantic representation, which happens to match the current discourse context, but which does not correspond to the user's expressed intent (e.g., user says 'Boston', but the speech recognizer returns 'Austin').

The ability to accurately detect such errors and recover from them is therefore paramount in any spoken language interface. Although certain error detection mechanisms can operate early in the input processing stage, other errors can only be detected at the dialog management level. Only at this level, after the system attempts to integrate the decoded semantics of the current user turn into the larger discourse context, enough information is accumulated to make a fully informed judgment in an error detection (or diagnosis) task. Similarly, the decisions to trigger various error handling strategies such as confirming a concept, asking the user to repeat, asking the user to rephrase, etc. are made at the dialog management level. The system has to balance the costs of engaging in an error recovery strategy against those of continuing the interaction with potentially incorrect information. Error handling is therefore a crucial function of any dialog manager.

In most traditional frameworks, the responsibility for handling potential errors is oftentimes delegated to the system developer. Error handling is therefore regarded as an integral part of the system's dialog task. In this case, the system developer would have to write code that handles potential misunderstanding and non-understanding errors. For instance, in VoiceXML, a developer needs to declare new <FIELD> blocks and the corresponding logic for implementing explicit and implicit confirmations (this has to be done for every slot separately). While still limited, there is somewhat more built-in support for handling non-understandings, via the <NOINPUT> and <NOMATCH> elements. This type of approach where the system developer is responsible for writing the error-handling code tends to lead to monolithic one-time solutions that lack portability and are difficult to maintain. Furthermore, it often results in inconsistent behaviors in different parts of the dialog.

The alternative is a decoupled, task-independent or "toolkit" (Mankoff et al., 2000) approach to error handling. Like Mankoff, we believe that, for a wide class of task-oriented systems, error handling can and should be regarded as a domain-independent conversational skill, and can therefore be decoupled from the domain-specific aspects of dialog control logic, i.e. from the dialog task specification. In general, a large number of error recovery strategies, such as asking the user to repeat or rephrase, are entirely task-independent. Other, more complex strategies such as explicit and implicit confirmations can also be decoupled from the task by using appropriate parameterizations. For instance, we can construct a generic "explicit confirmation for

concept *X*" conversational strategy, and use it repeatedly at runtime by parametrizing it accordingly (e.g., *X* = `departure_time` would lead to "Did you say you wanted to leave at 3pm?")

We argue that this type of approach is not only feasible in the context of a plan-based dialog management framework, but it also provides several important advantages. First, it increases the degree of consistency in the interaction style, both within and across tasks. This in turn leads to a better user experience, and facilitates the transference of learning effects across systems. Second, the approach significantly decreases development and maintenance efforts; it fosters reusability and ease-of-use. Ideally, system authors should not have to worry about handling understanding-errors. Rather, they should simply focus on describing the domain-specific dialog control logic, under the assumption that inputs to the system will always be perfect. At the same time, the dialog engine should automatically prevent understanding-errors or gracefully recover from them. Last but not least, a task-independent approach represents a more sensible choice from a software engineering perspective.

The case for separating out various task-independent aspects of the conversation has in fact been made previously. Balentine and Morgan (1999) recommend identifying a set of generic speech behaviors and constructing dialog systems starting from these basic building blocks. Lemon et al. (2003) propose a dialog management architecture where "content-level communicative processes" and "interaction-level phenomena" are handled in separate layers. The Universal Speech Interface project (Rosenfeld et al., 2000; Tomko, 2003; Tomko, 2007) proposes to identify and leverage a basic set of dialog universals that transfer across applications and domains. Error handling capabilities are also a good candidate for this type of decoupling, and can be implemented as domain- and task-independent conversational mechanisms.

In the rest of this section we describe the practical implementation for a decoupled, task-independent error handling architecture in the context of a complex, hierarchical plan-based dialog management framework. We begin with a high-level overview of the proposed error handling architecture.

4.3.1. Architectural overview

The error handling architecture in the RavenClaw dialog management framework subsumes two main components: (1) a set of error recovery strategies, and (2) an error handling decision process that triggers these strategies at the appropriate time – see Fig. 9.

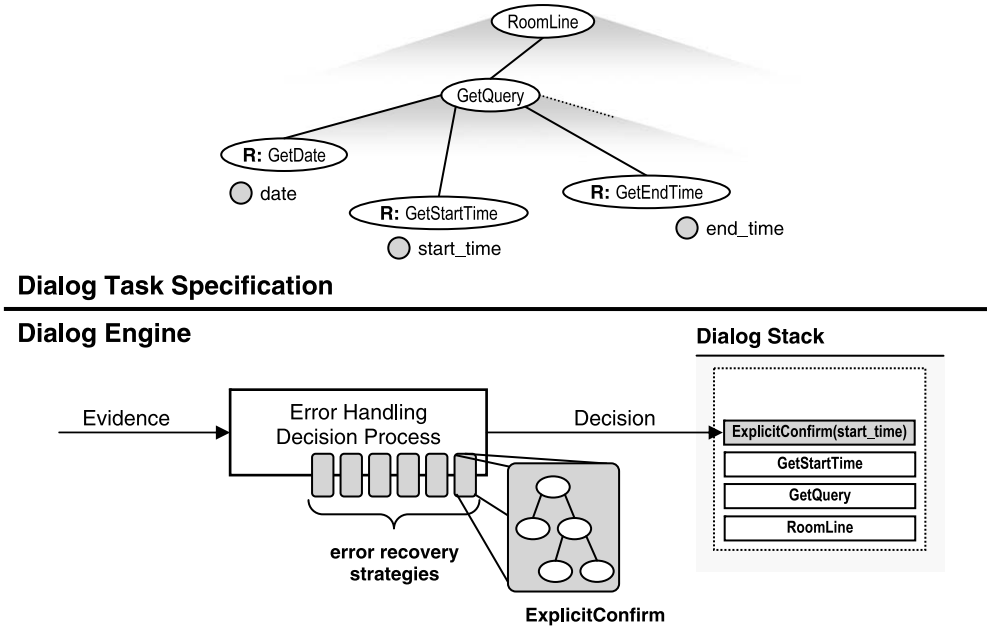

Fig. 9. RavenClaw error handling architecture – block diagram.

The error recovery strategies fall into two categories: (1) strategies for recovering from misunderstandings (e.g., explicit and implicit confirmation), and (2) strategies for recovering from non-understandings (e.g., asking the user to repeat, asking the user to rephrase, providing help, etc.). These strategies were authored using the RavenClaw dialog task specification formalism described earlier, and they are available as library dialog agencies. System authors simply specify which strategies should be used by the dialog manager, and configure them accordingly.

The responsibility for handling potential errors is delegated to the error handling decision process (EHDP in the sequel), a subcomponent of the RavenClaw dialog engine. During each execution phase, the EHDP collects available evidence and decides which error recovery strategy (if any) should be engaged. If action is deemed necessary, the EHDP creates an instance of the corresponding error recovery strategy, parameterizes it accordingly, and pushes it on the dialog stack. In effect the EHDP changes the dialog task to ensure that potential errors are handled accordingly. For instance, in the example illustrated in Fig. 9, based on the confidence score, the dialog engine decided to trigger an explicit confirmation for the `start_time` concept. The engine therefore instantiated an `ExplicitConfirm` dialog agency (which implements an explicit confirmation strategy), parameterized it by passing a pointer to the concept to be confirmed (in this case `start_time`), and placed it on the dialog stack. Next, the strategy executes. Once completed, it is removed from the stack and the dialog resumes from where it left off. During the execution of the explicit confirmation, all other dialog control mechanisms are still in place; for instance, the user could request more help, or even shift the current dialog topic.

This design, in which both the error recovery strategies and the EHDP are decoupled from each other as well as from the actual dialog task specification has a number of benefits. First, it significantly lessens the system development effort. System authors are free to write the dialog task specification (i.e., the domain-specific aspects of the dialog-control logic) under the assumption that the inputs to the system will always be understood correctly. The responsibility for ensuring that the system maintains accurate information and that the dialog advances normally towards its goals is delegated to the EHDP in the RavenClaw dialog engine. The EHDP will modify the dialog task dynamically, engaging various strategies to prevent and recover from errors. Second, the proposed architecture promotes the reusability of error handling strategies across different systems. A large set of error recovery strategies are currently available in the RavenClaw dialog management framework. These strategies, together with any new strategies developed by a system author can be easily plugged into any new or existing RavenClaw-based spoken dialog system. Lastly, the approach ensures uniformity and consistency in the system's behavior, both within and across systems.

4.3.2. Error recovery strategies

Based on the type of problem they address, the error recovery strategies in the RavenClaw dialog management framework can be divided into two groups: (1) strategies for handling potential misunderstandings and (2) strategies for handling non-understandings. The current set of strategies is illustrated in Table 1.

Two strategies are available for handling potential misunderstandings: explicit confirmation and implicit confirmation. In an **explicit confirmation**, the system asks the user a yes/no question, in an effort to directly (or explicitly) confirm a certain concept value. For instance: "Did you say you wanted a room starting at 10am?" – see Table 1. Alternatively, in an **implicit confirmation**, the system echoes back the value it heard to the user, and continues with the next question – "starting at 10am ... until what time?" The assumption is that, if the value is incorrect, the user will detect the error and interject a correction. In addition, although not currently present, other strategies for recovering from misunderstandings, such as disambiguation[3] can also be implemented and used in the RavenClaw dialog management framework.

A richer repertoire of error recovery strategies is available for dealing with non-understandings (see Table 1). An in-depth analysis of these strategies and their relative tradeoffs is available in Bohus and Rudnicky (2005b).

[3] In a disambiguation the system tries to determine which one of two concept hypotheses is the correct one. For instance: "Did you say Boston or Austin?"

Table 1
Task-independent error handling strategies in the RavenClaw dialog management system name.

Error Handling Strategy	Example
Strategies for handling misunderstandings	
Explicit confirmation	Did you say you wanted a room starting at 10 a.m.?
Implicit confirmation	starting at 10 a.m. … until what time?
Strategies for handling non-understandings [suppose a non-understanding happens after the system asks: "Would you like a small room or a large one?"]	
Notify that a non-understanding happened	Sorry, I didn't catch that …
Ask user to repeat	Can you please repeat that?
Ask user to rephrase	Can you please rephrase that?
Repeat prompt	Would you like a small room or a large one?
Give a you-can-say help message	For instance, you could say something like "I want a small room", or "I want a large room"
Give an explain-more help message	Right now I need you to tell me if you would prefer a small room or a large room.
Give a full help message	I found seven rooms available Friday from 10 to 12. Right now I need you to tell me if you would prefer a small room or a large room. For instance, you could say something like "I want a small room", or "I want a large room"
Give tips about how to best interact with the system	Okay, I know this conversation isn't going well. There are things you can try to help me understand you better. Speak clearly and naturally; don't speak too quickly or too slowly. Give short, concise answers. Calling from a quiet place helps. If you'd like to start from scratch, you can say 'start-over' at any time.
Ask for a short answer and repeat prompt	Please use shorter answers because I have trouble understanding long sentences… Would you like a small room or a large one?
Ask for a short answer and give a you-can-say help message	Please use shorter answers because I have trouble understanding long sentences… For instance, you could say something like "I want a small room", or "I want a large room"
Ask user to speak less loud and repeat prompt	I understand people best when they speak softer. Would you like a small room or a large one?
Fail the current request and move-on	Sorry, I didn't catch that. One choice would be Newell Simon 1507. This room can accommodate 50 people, and has a projector, a whiteboard and network access. Do you want a reservation for Newell Simon 1507?
Yield the turn	[…] (system remains silent, yielding the turn to the user)
Ask user if they'd like to start over	I'm sorry I'm still having trouble understanding you, and I might do better if we restarted. Would you like to start over?
Give up	I'm sorry but I'm having lots of trouble understanding you and I don't think I will be able to help you. Please call back during normal business hours.

4.3.3. Distributed error handling decision process

The decision of engaging one of the error recovery strategies described in the previous section is made by the **error handling decision process** in the RavenClaw dialog engine. This process is implemented in a distributed fashion. It consists of a collection of smaller **error handling models**, automatically associated with each request agent and each concept in the dialog task tree, as illustrated in Fig. 10.

The error handling models associated with individual concepts, also known as **concept error handling models**, are responsible for recovering from misunderstandings on those concepts. They use as evidence confidence scores for that particular concept (i.e., the current belief over that concept) and trigger the misunderstanding recovery strategies such as explicit or implicit confirmation.

The error handling models associated with individual request-agents, also known as **request error handling models**, are responsible for recovering from non-understandings that occur during the corresponding requests. They use as evidence features characterizing the current non-understanding and dialog state and trigger the

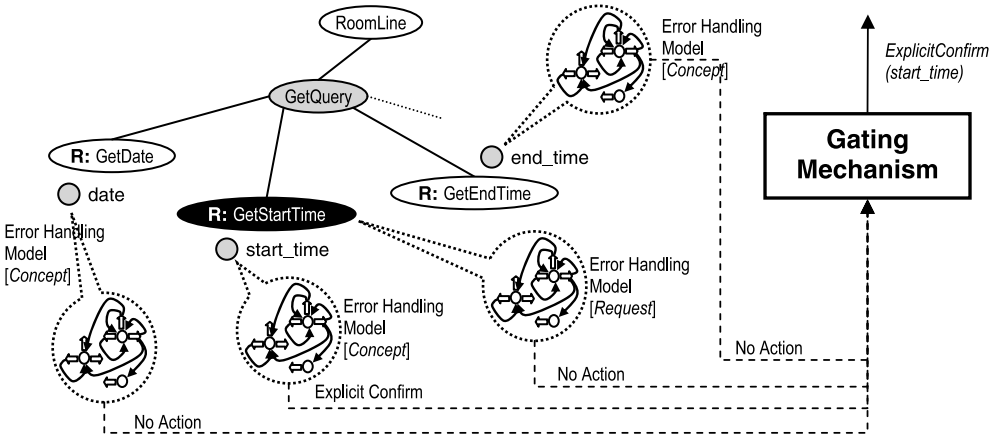

Fig. 10. Distributed error handling decision process.

non-understanding recovery strategies, such as asking the user to repeat, asking the user to rephrase, repeating the system prompt, providing help, etc.

During the error handling phase, each concept- and request error handling model computes and forwards its decision to a gating mechanism. The gating mechanism queues up these actions (if necessary) and executes them one at a time. For instance, in the example from Fig. 10, the error handling model for the `start_time` concept signaled that it wanted to trigger an explicit confirmation on that concept; the other local models did not take any action. In this case the gating mechanism created a new instance of an explicit confirmation agency, passed it a pointer to the concept to be confirmed (`start_time`), and placed it on the dialog stack, as illustrated in Fig. 9. On completion, the belief over the confirmed concept is updated in light of the user response, and the dialog resumes from where it left off.

The behavior of each error handling model is specified by means of a **control-policy**. Typically, a **control-policy** defines the model's behavior via a set of parameters that describe the cost of various actions under various circumstances. For instance a predefined "pessimistic" policy for a concept-level error handling model specifies a high cost for false-acceptance errors; as a result the model tends to always engage in explicit confirmations. Alternatively a predefined "optimistic" policy specifies a lower relative cost for false-acceptance errors, and as a result the model engages in explicit confirmations only if the confidence for the top concept hypothesis is below a certain threshold. The policies mentioned above are only two examples of simple predefined error handling policies available in the RavenClaw dialog management framework. A number of additional policies are available, and yet others can be easily created and used by the system author. The framework allows for a high degree of flexibility in the error handling behaviors. For instance, an error handling model might take into account the number of corrections previously made by the user and in effect create a dynamic error handling policy (i.e., the system responds differently based on the number of previous user corrections). Moreover, data-driven error handling models (i.e. using machine-learned policies) can and have also been implemented – see for instance our work on online supervised learning of non-understanding recovery policies (Bohus et al., 2006).

The distributed and encapsulated nature of the error handling decision increases its scalability and supports learning-based approaches to error handling. First, the structure and parameters of individual error handling models can be tied across different concepts or request-agents. In the example from Fig. 10, the error handling model for the `start_time` concept can be assumed to be identical to the one for the `end_time` concept; all models controlling Boolean (yes/no) concepts could be also tied together (e.g., the system author can configure at design-time all yes/no concepts in the task tree to use the same parameter set). Parameter tying can greatly improve the scalability of learning-based approaches because the data is polled together and the total number of parameters grows sub-linearly with the size of the task (i.e., with the number of concepts and request-agents in the dialog task tree). Secondly, policies learned in a system can be reused in other systems because the error handling models are decoupled from the actual dialog task specification. For instance, we expect that the grounding of yes/no concepts functions similarly at different locations in the dialog, but also across domains. Thirdly, the proposed architecture accommodates dynamic task generation. The dialog task tree (the dialog

plan) can be dynamically expanded at runtime, and the corresponding concept- and request- error handling models will be created on the fly. If the model structure and parameters are tied, we do not need to know the full structure of the task to be executed by the system in advance.

These advantages stem from an independence assumption made by the proposed architecture: the error handling models associated with different concepts and request-agents in the dialog task tree operate independently of each other. We believe that in practice the advantages gained by making this independence assumption and resorting to a distributed and encapsulated error handling process significantly outweigh the potential drawbacks.

4.4. Other domain-independent conversational strategies

Apart from the error handling strategies, the RavenClaw dialog management framework provides automatic support for a number of additional domain-independent conversational strategies. Examples include the ability to handle timeouts, requests for help, for repeating the last utterance, suspending and resuming the conversation, starting over, re-establishing the context, etc.

Internally, these strategies are implemented as library dialog agencies, using the dialog task specification formalism described earlier, in Section 4.1. The system author simply specifies which strategies the dialog engine should be using, and parameterizes them accordingly. The strategies are invoked automatically by the dialog engine at appropriate times. Typically, these strategies implement triggers that will automatically bring them into focus when the user makes a specific request. For instance, the `Repeat` strategy defines a trigger on the `[Repeat]` semantic grammar concept. When the user asks the system to repeat, the trigger fires and the dialog engine automatically places the `Repeat` strategy on the dialog stack. The strategy taps into the output management subcomponent in RavenClaw and repeats the last system request or utterance. The dialog then resumes from the point where it was left off.

In addition, system developers can write new task-independent conversational strategies, encapsulate them as library agents, and make them available for use in other RavenClaw-based dialog systems. The current architecture promotes reusability, and ensures consistency in behaviors both within and across systems.

5. RavenClaw-based systems

In the previous section, we described in detail the RavenClaw dialog management framework, and the algorithms and data structures that govern its function. We now turn our attention towards the problem of evaluation, and discuss several spoken dialog systems that have been developed to date using this framework.

Evaluation of spoken language interfaces is a difficult task, and has received significant amounts of attention from the research community (Walker et al., 1997; Walker et al., 1998; Hone and Graham, 2000; Walker et al., 2001; Hartikainen et al., 2004). Evaluating a dialog management framework poses even harder challenges. To our knowledge, no such objective assessments have been performed to date. Characteristics such as ease-of-use, portability, domain-independence, scalability, robustness, etc. are very hard to capture in a quantitative manner. In a sense, perhaps the problem is ill-posed. Comparing dialog management frameworks is like comparing programming languages: while arguments can be made about various strengths and weaknesses of various programming languages, no clear order or measure of absolute performance can be established. Certain programming languages are more appropriate for certain tasks than others. Even evaluating the suitability of a programming language (or dialog management framework) for a predefined task can be difficult, given that many applications can be recast into a form that is tractable in a particular approach.

As a first step towards a more rigorous evaluation of the RavenClaw dialog management framework, we decided to use this framework to build a number of spoken dialog systems spanning different domains and interaction types. In the process, we monitored various aspects of the development process and noted the degree of accommodation required by RavenClaw.

We have seen in the previous section that RavenClaw is a two-tier architecture that decouples the domain-specific aspects of the dialog control logic from the domain-independent dialog engine. To build a new dialog manager, system authors have to develop a dialog task specification, essentially a hierarchical plan for the interaction. The dialog engine manages the dialog by executing this dialog task specification. To date, about

Table 2
RavenClaw-based spoken dialog systems (highlighted systems are described in more detail in the text).

System name	Domain / Description	Interaction type
RoomLine	telephone-based system that provides support for conference room reservation and scheduling within the School of Computer Science at CMU.	information access (mixed initiative)
Let's Go! Public (Raux and Eskenazi 2004; Raux et al. 2005; Raux et al. 2006)	telephone-based system that provides access to bus route and scheduling information in the greater Pittsburgh area	information access (system initiative)
LARRI (Bohus and Rudnicky 2002)	multi-modal system that provides assistance to F/A-18 aircraft personnel during the execution of maintenance tasks	multi-modal task guidance and procedure browsing (mixed initiative)
Intelligent Procedure Assistant (Aist et al. 2003; Aist et al. 2004)	early prototype for a multi-modal system aimed at providing guidance and support to the astronauts on the International Space Station during the execution of procedural tasks and checklists	multi-modal task guidance and procedure browsing (mixed initiative)
TeamTalk (Harris et al. 2005)	multi-participant spoken language command-and-control interface for a team of robots operating in the treasure-hunt domain	multi-participant command-and-control (mixed initiative)
VERA	telephone-based taskable agent that can be instructed to deliver messages to a third party and make wake-up calls.	message-passing (system initiative)
Madeleine	text-based dialog system for medical diagnosis	medical diagnosis (system initiative)
ConQuest (Bohus, D. et al. 2007)	telephone-based spoken dialog system that provides conference schedule information (deployed during Interspeech-2006)	information access (mixed-initiative)

a dozen systems have been developed using the RavenClaw framework. Some of these systems have been successfully deployed into day-to-day use: Table 2 provides a quick summary. As this table shows, the systems operate in structurally different domains and were constructed by development teams with different degrees of experience building spoken language interfaces.

It is important to note that RavenClaw is a framework for developing dialog managers. However, in order to build a fully functioning spoken language interface, a number of other components besides a dialog manager are required: speech recognition, language understanding and generation, speech synthesis, etc. The systems described in Table 2 also rely on Olympus, a collection of freely available dialog system components.

We begin therefore by describing the Olympus dialog system infrastructure in the next subsection. Then, in the following four subsections, we briefly discuss four different spoken dialog system developed using the RavenClaw/Olympus framework. We present details about the domain, the interaction style, as well as various system characteristics such as vocabulary and grammar size, etc. In addition, we present performance statistics and discuss challenges encountered throughout development and deployment stages.

5.1. The Olympus dialog system infrastructure

Olympus (Bohus et al., 2007) is a dialog system infrastructure that, like RavenClaw, has its origins in the earlier CMU Communicator project (Rudnicky et al., 1999). At the high-level, Olympus implements a classical dialog system architecture, as previously illustrated in Fig. 1. Fig. 11 provides a more detailed account of a typical RavenClaw/Olympus based system. While the pipeline illustrated in Fig. 1 captures the logical flow of information in the system, in practice the various system components do not communicate directly. Rather, they rely on a centralized message-passing infrastructure – Galaxy (Seneff et al., 1998). Each component is implemented as a separate process that connects to a centralized traffic router – the Galaxy hub. The messages are sent through the hub, which forwards them to the appropriate destination. The routing logic is described by a configuration script.

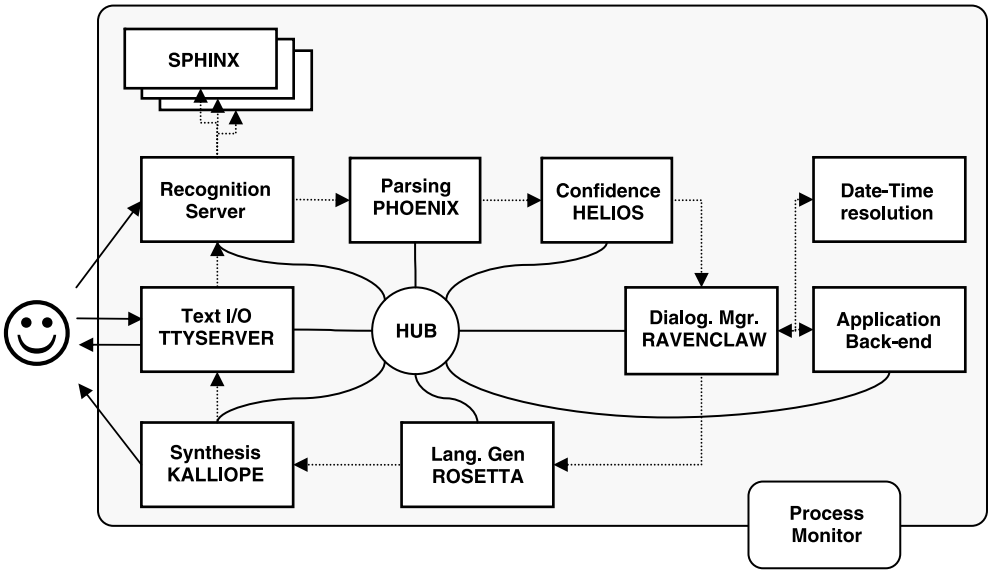

Fig. 11. The Olympus/RavenClaw dialog system architecture: a more detailed view.

For recognition, Olympus uses the Sphinx decoding engine (Huang et al., 1992). A recognition server component captures the audio stream (typically from the sound-card), forwards it to a set of parallel recognition engines, and collects the corresponding recognition results. The top-level recognition hypotheses (one from each engine) are then forwarded to the language understanding component. Currently, Sphinx-II (semi-continuous HMM recognition) and Sphinx-III (continuous HMM recognition) engines are available and can be used in conjunction with the recognition server. The individual recognition engines can be configured to use a variety of off-the-shelf acoustic models, and either n-gram or grammar-based language models. State-specific as well as class-based language models are supported, and tools for constructing language and acoustic models from data are readily available. Additionally, a DTMF (touch-tone) decoder is also available as a recognition engine. The RavenClaw/Olympus systems described in the next section use two parallel Sphinx-II recognizers: one configured with acoustic models trained using male speech and the other configured with acoustic models trained using female speech. Other parallel decoder configurations can also be created and used.

Language understanding is implemented via Phoenix, a robust semantic parser (Ward and Issar, 1994). Phoenix uses a semantic hand-written grammar to parse the incoming set of recognition hypotheses (one or more parses can be generated for each hypothesis). The semantic grammar is constructed by concatenating a set of reusable grammar rules that capture domain-independent constructs like `[Yes]`, `[No]`, `[Help]`, `[Repeat]`, `[Number]`, etc., with a set of domain-specific grammar rules authored by the system developer. For each recognition hypothesis the output of the parser consists of a sequence of slots containing the concepts extracted from the utterance.

From Phoenix, the set of parsed hypotheses is passed to Helios, the confidence annotation component. Helios uses features from different knowledge sources in the system (e.g., recognition, understanding, dialog, etc.) to compute a confidence score for each parsed hypothesis. This score reflects the probability of correct understanding, i.e. how much the system trusts that the current semantic interpretation corresponds to the user's expressed intent. The hypothesis with the highest confidence score is then forwarded to the dialog manager.

The next component in the chain is the RavenClaw-based dialog manager. The dialog manager integrates the semantic input in the current discourse context, and decides which action the system should engage in next. In the process, the dialog manager may consult/exchange information with a number of other domain-specific agents, such as an application-specific back-end.

The semantic output from the dialog manager is sent to the Rosetta language generation component, which creates the corresponding surface form. Rosetta supports template-based language generation. Like the grammar, the set of language generation templates is assembled by concatenating a set of predefined, domain-independent templates, with a set of manually authored task-specific templates.

Finally, the prompts are synthesized by the Kalliope speech synthesis module. Kalliope can be configured to use a variety of speech synthesis engines: Festival (Black and Lenzo, 2000), which is an open-source speech synthesis system, as well as Theta (Cepstral, 2005) and Swift (Cepstral, 2005), which are commercial solutions. Kalliope supports both open-domain (e.g. diphone) and limited-domain (e.g. unit selection) voices. The SSML markup language is also supported.

The various components briefly described above form the core of the Olympus dialog system infrastructure. Additional components have been added throughout the development of various systems, and, given the modularity of the Olympus architecture, they can be easily reused. Here are some examples. A text input-and-output server (TTYServer) provides support for text-based systems, and can also be very useful for debugging purposes. A GoogleTalk agent was implemented in the Olympus framework to support interactions via the popular internet messaging system. A Skype speech client component is also available. A process monitor component is used to start-up and to monitor all the other components in an Olympus system. Additionally, the process monitor can automatically restart components that crash, and send notification emails to a system administrator. Finally, a variety of logging and data processing and analysis tools are also available as part of the Olympus distribution.

We now briefly discuss four spoken dialog systems developed using the RavenClaw dialog management framework and the Olympus dialog infrastructure. We highlight the differences in the dialog domains for these systems, and discuss how the RavenClaw framework adapted to each of the specific requirements of each domain. Additional technical details regarding the other components in the systems (e.g. speech recognition, language understanding, etc.) are presented in Table 3.

Table 3
Implementation details about four RavenClaw based spoken dialog systems.

	RoomLine	Let's Go! Public	LARRI	TeamTalk
Recognition				
Engine	2 parallel Sphinx-II engines, with gender-specific acoustic models	3 parallel Sphinx-II engines with gender-specific acoustic models + touch-tone recognition model	single Sphinx-II decoder	single PocketSphinx decoder w/generic acoustic models
Vocabulary size	1092 words	7911 words	408 words	140 words
Language model	class-based bigram	state-specific, class-based trigram	WSJ-based trigram	class-based trigram
Avg.WER	26%	32%	not evaluated	not evaluated
Grammar				
Size	45 top-level slots	29 top-level slots	61 top-level slots	8 top-level slots
Dialog task				
# of dialog agents	117	62	dynamic (61 to several hundreds, depending on the procedural task)	125
# of concepts	28	17	dynamic	89
Error handling				
misunderstandings	explicit and implicit confirmations + confidence-based policy	explicit confirmations + pessimistic policy (always confirm)	explicit and implicit confirmations + confidence-based policy	explicit and implicit confirmations + confidence-based policy
non-understandings	10 recovery strategies	10 recovery strategies	5 recovery strategies	1 strategy (yield the turn)
Performance				
Task success	75%	76%	N/A	N/A

5.2. RoomLine

The first system we discuss is RoomLine, a telephone-based mixed-initiative spoken dialog system that provides conference room schedule information and allows users to make room reservations. The system has access to live information about the schedules of 13 conference rooms in two buildings on the CMU campus: Wean Hall and Newell Simon Hall. Additionally, the system has information about the various characteristics of these rooms such as location, size, network access, whiteboards, and audio-visual equipment. To perform a room reservation, the system finds the list of rooms that satisfy an initial set of user-specified constraints. Next, RoomLine presents this information to the user, and engages in a follow-up negotiation dialog to identify which room best matches the user's needs. Once the desired room is identified, registered users can authenticate using a 4-digit touch-tone PIN, and the system performs the reservation through the campus-wide CorporateTime calendar server. The system has been publically available to students on campus since 2003.

RoomLine therefore implements a classical slot-filling type interaction in the initial phase of the dialog, and a negotiation-type interaction in the second part – see sample conversations at RavenClaw-Olympus (2007). Both phases of the conversation are mixed-initiative. In the initial, information-gathering phase, the system asks a series of questions to determine the basic constraints, i.e., the date and time for the reservation. The user can however take the initiative at any point and specify additional constraints, such as a/v equipment, room size, etc. In the second phase of the conversation the system presents the list of rooms to the user, depending on the number of available rooms, it either asks a set of additional questions to narrow down the list, or starts suggesting rooms from the list. At the same time, the users can take the initiative and navigate the solution space by specifying additional constraints, or relaxing the existing ones.

The system uses both explicit and implicit confirmations to recover from potential misunderstandings, in conjunction with a confidence-based policy (e.g. explicitly confirm for high-confidence, implicitly confirm for medium-confidence and reject for low-confidence). To deal with non-understanding, the system uses ten non-understanding recovery strategies (see Table 1), in conjunction with a predefined heuristic policy.

The plan-based RavenClaw dialog management framework adapted easily to this domain and fully supports both the information gathering and the negotiation interaction-types. The expectation agenda input processing allows the system to incorporate additional constraints specified by the user at any time, and the task-specific dialog logic is captured via a combination of triggers, preconditions and effects on the various agents in the dialog task tree.

5.3. Let's Go! (Public) Bus Information System

The second system we discuss is the Let's Go! Public Bus Information System (Raux and Eskenazi, 2004; Raux et al., 2005; Raux et al., 2006), a telephone-based system that provides access to bus route and schedule information. The system knows about 12 bus routes, and 1800 place names in the greater Pittsburgh area. In order to provide bus schedule information, the system tries to identify the user's departure and arrival stop, and the departure or arrival time. Once the results are provided, the user can ask for the next or previous bus on that route, or can restart the conversation from the beginning to get information for a different route. A sample interaction with this system is presented in RavenClaw-Olympus (2007).

Like RoomLine, Let's Go! Public is a slot-filling, information-access system. The conversation begins with an open-ended "How can I help you?" prompt, but continues with a set of focused questions in which the system asks in order for the departure place, arrival place and travel time. Again, the user is allowed to over-answer any particular question, but all concept values are explicitly confirmed by the system before moving on – this is simply accomplished by using a pessimistic confirmation policy on the relevant concepts in the dialog task tree. The decision to use this more conservative error handling strategy was based on the nature of the domain and interactions: real-users calling from high-noise environments (e.g. street). After the results are presented to the user, the system offers a menu-style set of options for finding more information about the current selected route or restarting the conversation. Like with RoomLine, no significant challenges were encountered in implementing this type of interaction using the RavenClaw dialog management framework.

In fall of 2004 the management of the Port Authority of Allegheny County (PAAC) found that the system could correspond to their user's needs. After a redesign stage aimed to increase robustness, the system was

open to the Pittsburgh population on March 4th, 2005. The system is connected to the PAAC customer service line during non-business hours, when no operators are available to answer the calls (i.e., 7 pm to 6 am on weekdays, and 6 pm to 8 am on weekends and national holidays). Since it deployment, the system has serviced on average about 40–50 calls per night, for a total of over 30,000 calls. Additional information is also available on the system's web-site (Go, 2008).

5.4. LARRI

LARRI (Bohus and Rudnicky, 2002), or the Language Based Retrieval of Repair Information system is a multi-modal system for support of maintenance and repair activities for F/A-18 aircraft mechanics. The system implements a level 4/5 Interactive Electronic Technical Manual (IETM), that is, semantically annotated documentation. LARRI integrates a graphical user interfaces for easy visualization of dense technical information (e.g. instructions, schematics, video-streams, etc.) with a spoken dialog system that facilitates information access and offers assistance throughout the execution of procedural tasks.

After the user logs into the system, LARRI retrieves the user's profile from a backend agent, and allows the user to view their current schedule and to select a task to be executed. The typical maintenance task consists of a sequence of steps, which contain instructions, optionally followed by verification questions in which the possible outcomes of each step are discussed. Basic steps can also include animations or short video sequences that can be accessed by the user through the GUI or through spoken commands. By default, LARRI guides the user through the procedure, in a step-by-step fashion. At the same time, the user can take the initiative and perform random access to the documentation/task, either by accessing the GUI or by simple spoken language commands such as *"go to step 15*" or *"show me the figure*". Once the maintenance task is completed, the system provides a brief summary of the activity, updates the information on the back-end side, and moves to the next scheduled task. The error handling configuration is similar to RoomLine: the system uses both explicit and implicit confirmations with a confidence-based policy, and a sub-set of five non-understanding recovery strategies. A sample interaction with the system is available at RavenClaw-Olympus (2007).

In contrast to the previous two systems we have described, which operate in information-access domains, LARRI provides assistance and guidance to the user throughout the execution of a procedural task. The hierarchical representation of the dialog task used in RavenClaw is very well suited in this domain, as it can be mapped directly onto the structure of the actual tasks to be performed by the user (with sub-tasks, steps, sub-steps, etc.). Moreover, since the procedural tasks are extracted on-the-fly from a task repository, the framework's ability to dynamically generate/expand the dialog task at runtime plays a very important role. The authoring of the dialog plan is therefore generic, i.e. independent of the specifics of the procedural tasks to be performed by the user. Once the user specifies the task to be performed, the procedure is loaded from the backend and a corresponding dialog task tree is generated. The number of agents and concepts in the dialog task can grow from a minimum of 61 to several hundred, depending on size of the current procedure. The Intelligent Procedure Assistant (Aist et al., 2003; Aist et al., 2004) and Madeleine (see Table 2) are two other RavenClaw-based systems where the dialog task is dynamically generated. The first system is very similar to LARRI, in that it is aimed at providing guidance and assistance through procedural tasks to astronauts on the international space station. In the second system, Madeleine, the dialog task is also dynamically constructed, this time based on medical diagnosis trees loaded at runtime from the backend. In practice, we found that RavenClaw framework was adapted easily and scaled up gracefully with the size of the task each of these cases.

A second important difference to the previously discussed systems is that LARRI is a multimodal application that integrates a graphical user interface for easy visualization of dense technical information (e.g. instructions, video-sequences, animations, etc.) with a spoken language interface that facilitates information access and offers task guidance in this environment. The graphical interface is accessible via a translucent head-worn display connected to a wearable client computer. A rotary mouse (dial) provides direct access to the GUI elements. The GUI is connected via the Galaxy hub to the rest of the system: the rotary mouse events are rendered as semantic inputs and are sent to Helios which performs the multimodal integration and forwards the corresponding messages to the dialog manager. The dialog manager therefore treats equivalent inputs received on different channels in a uniform fashion; the dialog authoring effort is similar to that of a speech-only system.

Although this system was never deployed into day-to-day use, we did evaluate LARRI on two separate occasions. The evaluations were performed on a military base, with the participation of trained Navy personnel, and were focused on understanding the experience of the mechanics. While users commented favorably on the language-based interface, a closer analysis of the sessions and feedback revealed a number of issues. They included the need for better feedback on the system's state; a more flexible (controllable) balance between the spoken and graphical outputs, as well as improvements to the GUI design. For the interested reader, a more detailed discussion is available in Bohus and Rudnicky (2002).

5.5. TeamTalk

TeamTalk (Harris et al., 2005; Dias et al., 2006) is a multi-participant spoken language interface that facilitates communication between a human operator and a team of robots. The system operates in a multi-robot-assisted treasure-hunt domain. The human operator is tasked to search a space for objects of interest and to bring those objects to a specified location. To achieve this task, the human operator directs the robots in a search operation. For instance, in one experiment (Harris et al., 2005), users were required to navigate (only by using the speech channel) two robots through corridors towards a certain location in a building.

The domain is command-and-control in flavor, but poses a number of additional challenges due to the multi-participant nature of the conversation. On the input side, the robots need to be able to identify who the addressee of any given user utterance is. On the output side, the robots need to address the problem of channel contention (i.e., multiple participants speaking over each other). For the interested reader, details about the current solutions to these problems are discussed in Harris et al. (2005). More generally, apart from the multi-participant aspects, some of the other research goals in the TeamTalk project are: (1) understanding the skills needed for communication in a human-robot team, (2) developing languages for robot navigation in novel environments and (3) understand how novel objects, locations and tasks come to be described in language.

The RavenClaw/Olympus framework was also relatively easily adapted to the demands of this domain. In the current architecture, each robot uses its own RavenClaw-based dialog manager, but all robots share the other Olympus components: Sphinx-II based speech recognition, Phoenix-based language understanding, Rosetta-based language generation and Festival-based speech synthesis (each robot uses a different voice). TeamTalk can interface with real robots, including the Pioneer P2DX and the Segway RMP. In addition, it can interface with virtual robots within the high-fidelity USARSim (Balakirsky et al., 2006) simulation environment. The processed user inputs are sent to all dialog managers (robots) in the system; each dialog manager decides based on a simple algorithm (Harris et al., 2005) whether or not the current input is addressed to it. If so, an action is taken; otherwise the input is ignored (it will be processed and responded to by another robot, i.e., dialog manager).

Only minor changes were required in the RavenClaw/Olympus architecture to support multiple dialog managers. For instance, the RavenClaw output messages were augmented with information about the identity of the dialog manager that generated them; this information was later used by the synthesis component to decide on the voice to use.

6. Discussion and future work

In this paper, we have described RavenClaw, a plan-based, task-independent dialog management framework. The framework finds its roots in the earlier Agenda dialog management architecture used in the CMU Communicator project (Rudnicky et al., 1999). In contrast to the ad-hoc tree-of-handlers used in the Agenda architecture (which captured the entire dialog task), RavenClaw enforces a clear separation between the domain-specific aspects of the dialog control logic and domain-independent conversational strategies. The domain-specific aspects are captured via a hierarchical plan for the conversation, constructed by the system author. A domain-independent dialog engine uses a stack to track the discourse structure and plans the conversation based on the specified dialog logic and inputs received from the user. The inputs are integrated into the discourse by means of a specialized data structure – the expectation agenda, which provides support for mixed-initiative interaction, context-based disambiguation and in general affords fine-grained control over

input processing (both the dialog and lower input processing levels – e.g. dynamic state-specific language modeling). Furthermore, the RavenClaw dialog engine automatically provides and controls a rich repertoire of reusable conversational skills, such as error handling, timing and turn-taking, context establishment, etc.

A number of parallels may be drawn between RavenClaw and other dialog management frameworks. Architecturally, RavenClaw is perhaps most similar to Collagen (Rich and Sidner, 1998; Rich et al., 2001). Albeit developed independently, both frameworks use a hierarchical plan to represent the interaction (a collection of "recipes" in Collagen), and a stack to track the nested structure of discourse at runtime. The expectation agenda data-structure and its affordances are unique to the RavenClaw framework. Parallels may also be drawn between RavenClaw and VoiceXML, the industry standard for development of voice-enabled applications. For instance, the agencies in RavenClaw can be seen as loosely corresponding to <FORM> elements in VoiceXML, the inform agents to <BLOCK> elements, and the request agents to <FIELD> elements. Some similarities can also be found between the expectation agenda and the scopes of grammars on different levels in VoiceXML (application, form and field). The full-blown hierarchical plan representation of the domain-specific control logic, coupled with the finer-grained control over input processing afforded by the expectation agenda creates however a significantly higher degree of versatility and scalability in the RavenClaw dialog management framework. In addition, the task-decoupled implementation and control of generic conversational skills like error handling further lessens the system development and maintenance efforts.

To date, over a dozen spoken dialog systems have been developed using the RavenClaw dialog management framework. These systems operate across a variety of domains, including information-access, guidance through procedural tasks, command-and-control, and message delivery. Three of these systems – RoomLine, Let's Go! Public, and ConQuest, have been successfully deployed into daily use. As we have seen in the previous section, the RavenClaw dialog management framework has easily adapted to the specific characteristics and requirements of each of these domains.

In developing these systems, we have learned that the high degree of flexibility afforded by the hierarchical plan-based dialog task specification can be very beneficial for developing complex interactions. Consider for instance the less flexible finite-state paradigm for designing and specifying an interaction plan. In this case, the interaction plan is specified in a positive or explicit manner, by specifying possible transitions in the interaction: for instance, from state A the system can move to states B, C or D. Developing complex, mixed-initiative interactions in this formalism is in general a labor-intensive task: the number of possible transitions increases quickly with the total number of states. In contrast, in RavenClaw, the interaction plan is by default[4] defined in a negative or implicit fashion, by specifying logical constraints that need to always hold. The flow control is therefore underspecified, and many different paths through the dialog task may be taken by the dialog planner, as long as the constraints expressed by the developer are satisfied. At runtime, a specific path is generated, depending on the user inputs and on their interplay with the system's conversational competencies. The plan-based representation therefore allows for easily coding mixed-initiative interaction and lessens development effort for complex systems, including cases where the dialog task is constructed dynamically at runtime (e.g. LARRI, the Intelligent Procedure Assistant and Madeleine).

At the same time, we have also found that the increased expressive power of this representation if not properly harnessed, can bring with it some unexpected challenges. Given the flexibility of the dialog planner, debugging the system or tracing the reasoning behind certain dialog control decisions and outcomes becomes at times a more difficult task. To alleviate these problems, an interesting direction to consider for future research is exploring more closely the spectrum of possibilities between the two polarities we have mentioned above: the constraint-based task representation and the explicit state transition representation.

Another important characteristic of the RavenClaw dialog management framework is that it decouples the domain-specific aspects of the dialog control logic from domain-independent conversational skills, such as error handling, providing help, re-establishing the context after a sub-dialog, supporting requests for repeating the last utterance, starting-over the conversation, etc. In combination with the plan-based task representation,

[4] Note that the framework allows the system developer to implement a different planning mechanism, including finite-state control in any of the agencies in the dialog task tree; this can be accomplished by overwriting the Execute routine for the desired agencies.

this decoupled approach confers good scalability properties, supports dynamic generation of dialog tasks[5], and favors learning based approaches for engaging such strategies (Bohus and Rudnicky, 2005b; Bohus et al., 2006; Bohus and Rudnicky, 2006). The error handling problems are encapsulated and solved locally and decoupled from the actual task performed by the system. Parameters for different local error handling models can be tied across models; this can improve scalability of learning-based approaches, because the total number of parameters to be learned grows sub-linearly with the size of the dialog task. New (error recovery) conversational strategies can be implemented and easily integrated into any existing system (Raux et al., 2006). Changing the system's behavior from an optimistic and to a pessimistic confirmation policy can be done by modifying a single parameter (and can be accomplished just as easily at design- or at run-time).

At the same time, the distributed error handling process still has a couple of limitations. For instance, long-range dependencies (e.g. inter-concept relationships) are not directly captured; in some situations, these dependencies can provide additional useful information for the error handling process. For instance, knowing that the start-time for a room reservation is 2 pm puts a number of constraints on what the end-time can be. Furthermore, the error handling strategies operate individually on concepts or request agents. The gating mechanism implements a heuristic that can sometimes couple multiple recovery actions in a single turn. For instance an implicit confirmation can be coupled with a follow-up explicit confirmation on a different concept: "a small room . . .. Did you say in Wean Hall?" Difficulties arise however if we want the system to combine multiple concepts in a single confirmation action, such as "Did you say you wanted a small room in Wean Hall?" The view taken so far in the RavenClaw dialog management framework is that, to a large extent, error handling can be modeled as a local process. In future work it will be interesting to explore in-depth and address some of the limitations described above.

Other interesting directions to be explored in future work regard further uses of the structure of the expectation agenda. As we have described earlier, in Section 4.2.2.4, the expectation agenda could be used to construct dynamically-interpolated language models. These models would take into account the current context and dialog history in a more accurate manner than a simple state-specific language model and might lead to further performance improvements. Another interesting direction is building an automatic, perhaps domain-independent method for controlling the depth of bindings in the expectation agenda (leading to an automatic adjustment of the level of initiative the system allows the user to take).

Two other important dialog phenomena we have not touched upon in this paper are anaphora and ellipsis. The view taken in this work is that the dialog manager is responsible for planning the high-level dialog actions, based on semantic inputs received from the lower stages in the input pipeline. We assume that anaphora and ellipsis have been resolved by the time the semantic input reaches the dialog manager. While these mechanisms can use information gathered at the dialog control level (e.g. history, current discourse objects, etc.), our view is that they are not a direct part of the dialog management task, but rather they belong in the earlier stages of the input processing pipeline. In practice, no components for anaphora or ellipsis resolution are currently available in the RavenClaw/Olympus framework (they have not been necessary in the systems and domains developed to date). Moving forward however, we expect that in more complex domains (e.g. tutoring), these phenomena play a more substantial role and future work will need to address this limitation.

Like the majority of other dialog management frameworks, the current version of RavenClaw makes a rigid "one-speaker-at-a-time" assumption. Although barge-in capabilities are supported, the dialog engine works asynchronously from the real world: it does not use low-level information about the realization of various utterances; rather, utterances and actions are assumed to be executed instantaneously, as soon as they are planned. User barge-ins and backchannels can therefore be interpreted in an incorrect context and can lead to turn overtaking problems, and sometimes even to complete breakdowns in the interaction. The RavenClaw framework and surrounding Olympus infrastructure are currently being updated to enable better, real-time reactive behaviors and more robust timing and turn-taking (Raux and Eskenazi, 2007). This updated architecture has already been deployed in the latest version of the Let's Go! Public spoken dialog system.

Finally, the TeamTalk project, led by Harris et al. (2005) aims to extend even further the capabilities of the RavenClaw dialog management framework in the direction of multi-participant conversation. Most dialog

[5] The dialog task tree can grow dynamically at runtime; the corresponding concept and request-agent error handling models will be automatically created by the dialog engine.

systems are built for one-to-one conversation. However, new issues arise once we move to a multi-participant setting, where many agents can be simultaneously engaged in a conversation. For instance, a number of problems regarding the conversation floor and the threading of sub-dialogs must be solved: how does each system identify who has the conversation floor and who is the addressee for any spoken utterance? How can multiple agents solve the channel contention problem, i.e., multiple agents speaking over each other?

Acknowledgement

We would like to thank Antoine Raux who has been a major contributor in the development of the RavenClaw dialog management framework and of the Olympus dialog system infrastructure. In addition we would also like to thank Svetlana Stenchikova, Jahanzeb Sherwani, Alan Black, Maxine Eskenazi, Gregory Aist, Satanjeev Banerjee, Stefanie Tomko, Ananlada Chotimongkol, and the students from the Dialogs on Dialogs student reading group for valuable help and discussions. Finally, we would also like to thank the anonymous reviewers for useful comments and feedback.

References

Aist, G., Dowding, J., Hockey, B.A., Rayner, M., Hieronymus, J., Bohus, D., Boven, B., Blaylock, N., Campana, E., Early, S., Gorrell, G., Phan, S., 2003. Talking through procedures: an intelligent Space Station procedure assistant. In: Proceedings of EACL-2003, Budapest, Hungary.

Aist, G., Bohus, D., Boven, B., Campana, E., Early, S., Phan, S., 2004. Initial development of a voice-activated astronaut assistant for procedural tasks: from need to concept to prototype. Journal of Interactive Instruction Development 16 (3), 32–36.

Aust, H., Schroer, O., 1998. An overview of the Philips dialog system. In: Proceedings of DARPA Broadcast News Transcription and Understanding Workshop, Lans-downe, VA.

Balakirsky, S., Scrapper, C., Carpin, S., Lewis, M., 2006. UsarSim: providing a framework for multirobot performance evaluation. In: Proceedings of PerMIS.

Balentine, B., Morgan, D., 1999. How to Build a Speech Recognition Application: A Style for Telephony Dialogues. Enterprise Integration Group.

Black, A., Lenzo, K., 2000. Building Voices in the Festival Speech System, 2000. Available from: <http://festvox.org/bsv/>.

Bohus, D., 2007. On-line List of Spoken Dialog Systems, Retrieved January 2007. Available from: <http://www.cs.cmu.edu/~dbohus/SDS>.

Bohus, D., Rudnicky, A., 2002. LARRI: a language-based maintenance and repair assistant. In: Proceedings of ISCA Tutorial and Research Workshop on Multi-modal Dialogue in Mobile Environments (IDS'02), Kloster Irsee, Germany.

Bohus, D., Rudnicky, A., 2003. RavenClaw: dialog management using hierarchical task decomposition and an expectation agenda. In: Proceedings of Interspeech-2006, Geneva, Switzerland.

Bohus, D., Rudnicky, A., 2005a. Constructing accurate beliefs in spoken dialog systems. In: Proceedings of ASRU-2005, San Juan, Puerto Rico.

Bohus, D., Rudnicky, A., 2005b. Sorry, I didn't catch that! – an investigation of non-understanding errors and recovery strategies. In: Proceedings of 6th SIGdial Workshop on Discourse and Dialog (SIGdial'05), Lisbon, Portugal.

Bohus, D., Rudnicky, A., 2006. A K hypotheses + other belief updating model. In: Proceedings of AAAI Workshop on Stochastic Methods in Spoken Dialog Systems, Boston, MA.

Bohus, D., Langner, B., Raux, A., Black, A.W., Eskenazi, M., Rudnicky, A., 2006. online supervised learning of non-understanding recovery policies. In: Proceedings of SLT-2006, Palm Beach, Aruba.

Bohus, D., Puerto, S.G., Huggins-Daines, D., Keri, V., Krishna, G., Kumar, R., Raux, A., Tomko, S., 2007a. Conquest – an open-source dialog system for conferences. In: Proceedings of HLT'07, Rochester, NY.

Bohus, D., Raux, A., Harris, T., Eskenazi, M., Rudnicky, A., 2007b. Olympus: an open-source framework for conversational spoken language interface research. In: Proceedings of Bridging the Gap: Academic and Industrial Research in Dialog Technology Workshop, Rochester, NY.

Bos, J., Klein, E., Lemon, O., Oka, T., 2003. DIPPER: description and formalisation of an information-state update dialogue system architecture. In: Proceedings of SIGdial'03, Sapporo, Japan.

Cepstral, L., 2005. SwiftTM: small footprint text-to-speech synthesizer, 2005. Available from: <http://www.cepstral.com>.

Cole, R., 1999. Tools for research and education in speech science. In: Proceedings of International Conference of Phonetic Sciences, San Francisco, CA.

Dias, M.B., Harris, T.K., Browning, B., Jones, E.G., Argall, B., Veloso, M., Stenz, A., Rudnicky, A.I., 2006. Dynamically formed human-robot teams performing coordinated tasks. In: Proceedings of AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone.

Dowding, J., Gawron, J., Appelt, D., Cherny, L., Moore, R., Moran, D., 1993. Gemini: a natural language system for spoken language understanding. In: Proceedings of ACL'93, Columbus, OH.

Ferguson, G., Allen, J., 1998. TRIPS: an intelligent integrated problem-solving assistant. In: Proceedings of AAAI'98, Madison, WI.

Go, L.S., 2008. Let's Go Lab: Evaluation of Spoken Dialog Techniques with Real Users, 2008. Available from: <http://accent.speech.cs.cmu.edu/>.
Gruenstein, A., Wang, C., Seneff, S., 2005. Context-sensitive statistical language modeling. In: Proceedings of Interspeech-2005, Lisbon.
Gustafson, J., Bell, L., Beskow, J., Boye, J., Carlson, R., Edlund, J., Granstorm, B., House, D., Wiren, M., 2000. AdApt – a multimodal conversational dialogue system in an apartment domain. In: Proceedings of ICSLP'00, Beijing, China.
Harris, T., Banerjee, S., Rudnicky, A., 2005. Heterogeneous multi-robot dialogues for search tasks. In: Proceedings of AAAI Spring Symposium: Dialogical Robots, Palo Alto, CA.
Hartikainen, M., Salonen, E.-P., Turunen, M., 2004. Subjective evaluation of spoken dialogue systems using SER VQUAL method. In: Proceedings of ICSLP'04, Jeju Island, Korea.
Hone, K.S., Graham, R., 2000. Towards a tool for the subjective assessment of speech system interfaces (SASSI). Natural Language Engineering 6 (3/4), 287–305.
Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., Lee, K.-F., Rosenfeld, R., 1992. The SPHINX-II speech recognition system: an overview. Computer Speech and Language 7 (2), 137–148.
Larsson, S., 2002. Issue-based dialog management. Goteborg University. Ph.D.
Lemon, O., Gruenstein, A., Cavedon, L., Peters, S., 2002. Multi-tasking and collaborative activities in dialogue systems. In: Proceedings of SigDIAL'02, Philadelphia, PA.
Lemon, O., Cavedon, L., Kelly, B., 2003. Managing dialogue interaction: a multi-layered approach. In: Proceedings of SigDIAL'03, Sapporo, Japan.
Mankoff, J., Hudson, S., Abowd, G., 2000. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In: Proceedings of CHI-2000. The Hague, Amsterdam.
McTear, M., 2002. Spoken dialogue technology: enabling the conversational user interface. ACM Computing Surverys 34 (1), 90–169.
Paek, T., Horvitz, E., 2000. Conversation as action under uncertainty. In: Proceedings of Sixteenth Conference on Uncertainty and Artificial Intelligence, Stanford, CA.
Raux, A., Eskenazi, M., 2004. Non-native users in the let's go!! spoken dialogue system: dealing with linguistic mismatch. In: Proceedings of HLT-NAACL'04, Boston, MA.
Raux, A., Eskenazi, M., 2007. A multi-layer architecture for semi-synchronous event-driven dialogue management. In: Proceedings of ASRU-2007, Kyoto, Japan.
Raux, A., Langner, B., Bohus, D., Black, A., Eskenazi, M., 2005. Let's go public! taking a spoken dialog system to the real world. In: Proceedings of 9th European Conference on Speech Communication and Technology (Interspeech'05), Lisbon, Portugal.
Raux, A., Bohus, D., Langner, B., Black, A.W., Eskenazi, M., 2006. Doing research on a deployed spoken dialogue system: one year of let's go! experience. In: Proceedings of nineth International Conference on Spoken Language Processing (InterSpeech'06), Pittsburgh, PA.
RavenClaw-Olympus, 2007. RavenClaw-Olympus Web Page. Retrieved January 2007. Available from: <http://www.ravenclaw-olympus.org>.
Rich, C., Sidner, C., 1998. COLLAGEN: a collaboration manager for software interface agents. An International Journal: User Modeling and User–Adapted Interaction 8 (3/4), 315–350.
Rich, C., Sidner, C., Lesh, N., 2001. COLLAGEN: applying collaborative discourse theory to human–computer interaction. AI Magazine 22, 15–25.
Rosenfeld, R., Olsen, D., Rudnicky, A., 2000. A universal human–machine speech interface. Technical Report Pittsburgh, PA, Carnegie Mellon University.
Rudnicky, A., Thayer, E., Constantinides, P., Tchou, C., Stern, R., Lenzo, K., Xu, W., Oh, A., 1999. Creating natural dialogs in the Carnegie Mellon communicator system. In: Proceedings of 6th European Conference on Speech Communication and Technology (Eurospeech'99), Budapest, Hungary.
Seneff, S., Hurley, E., Lau, R., Pao, C., Schmid, P., Zue, V., 1998. Galaxy-II: a reference architecture for conversational system development. In: Proceedings of ICSLP'98, Sydney, Australia.
Swerts, M., Litman, D., Hirschberg, J., 2000. Corrections in spoken dialogue systems. In: Proceedings of ICSLP-2000, Beijing, China.
Tomko, S., 2003. Speech graffiti: assessing the user experience. Language Technologies Institute, Carnegie Mellon University, Master.
Tomko, S., 2007. Improving interaction with spoken dialog systems via shaping. Language Technologies Institute, Pittsburgh, PA, Carnegie Mellon University, Ph.D.
TrindiKit, 2007. TrindiKit Retrieved March 2007. Available from: <http://www.ling.gu.se/projekt/trindi/trindikit/>.
Understanding, C.f.S.L.m 2007. CSLU Toolkit Retrieved March 2007. Available from: <http://cslu.cse.ogi.edu/toolkit/>.
VoiceXML, 2007. VoiceXML Specification, March 2007. Available from: <http://www.voicexml.org>.
Walker, M., Litman, D., Kamm, C., Abella, A., 1997. PARADISE: a general framework for evaluating spoken dialogue agents. In: Proceedings of ACL/EACL.
Walker, M., Litman, D., Kamm, C., Abella, A., 1998. Evaluating spoken dialogue systems with PARADISE: two case studies. Computer Speech and Language 12 (3).
Walker, M., Passonneau, R., Boland, J., 2001. Quantitative and qualitative evaluation of the DARPA communicator spoken dialogue systems. In: Proceedings of Annual Meeting of the Association for Computational Linguistics (ACL'01), Toulouse, France.
Ward, W., Issar, S., 1994. Recent improvements in the CMU spoken language understanding system. In: Proceedings of ARPA Human Language Technology Workshop, Plainsboro, NJ.
Xu, W., Rudnicky, A., 2000. Language modeling for dialogue systems. In: Proceedings of ICSLP'00, Beijing, China.
Zue, V., Seneff, S., Glass, J., Polifroni, J., Pao, C., Hazen, T., Hetherington, L., 2000. JUPITER: a telephone-based conversational interface for weather information. In: IEEE Transactions on Speech and Audio Processing 8(1).