The Curious Robot as a Case-Study for Comparing Dialog Systems

Julia Peltason and Britta Wrede

Abstract

Modeling interaction with robots raises new and different challenges for dialog modeling than traditional dialog modeling with less embodied machines. We present four case studies of implementing a typical human-robot interaction scenario with different state-of-the-art dialog frameworks in order to identify challenges and pitfalls specific to HRI and potential solutions. The results are discussed with a special focus on the interplay between dialog and task modeling on robots.

Introduction

The extension of the focus of research on robotics from vision-based physical interaction with the environment towards more social multi-modal, including speech-based, interactions with humans brings fundamentally new challenges to dialog modeling. Instead of designing interactions for information-oriented query systems – which have also been extended to virtual agents – it has now become necessary to take physical situatedness into account. This means that questions of reactability to dynamic environmens, possibly involving multiple modalities, and of potentially open-ended, unstructured interactions, infolving multiple tasks at a time play an important role and have to be considered in the dialog model. Also, the idea of the robot as a partner requires a mixed-initiative interaction style that enables both interaction partners to exchange information on their own initiative, to suggest new tasks, and ultimately the capability to learn from each other.

In light of these challenges, we have suggested an approach to dialog modeling on robots that – while keeping task and dialog structure well separated – tightly integrates dialog and domain level and includes concepts that support rapid prototyping of interaction scenarios which in our research has proven a valuable factor to boost dialog development through gathering wide spread experience and sampling data (Peltason and Wrede 2010b), (Peltason and Wrede 2010a).

In this article, we focus on the special demands that robot applications impose on the developer and compare our approach, called PaMini (for Pattern-Based Mixed-Initiative Human-Robot-Interaction), with existing, well-established dialog modeling approaches to identify problematic issues and potential remedies from different approaches (being well aware that most of them originally had not been intended for robotics).

The aim of this comparison is twofold. On the one hand, it is meant to give an overview of state-of-the-art dialog modeling techniques, and to illustrate the differences between these. On the other hand, we attempt to illustrate the differences between robotics and traditional domains for speech applications, and to point out potential pitfalls in robotics.

The target scenario: a Curious Robot

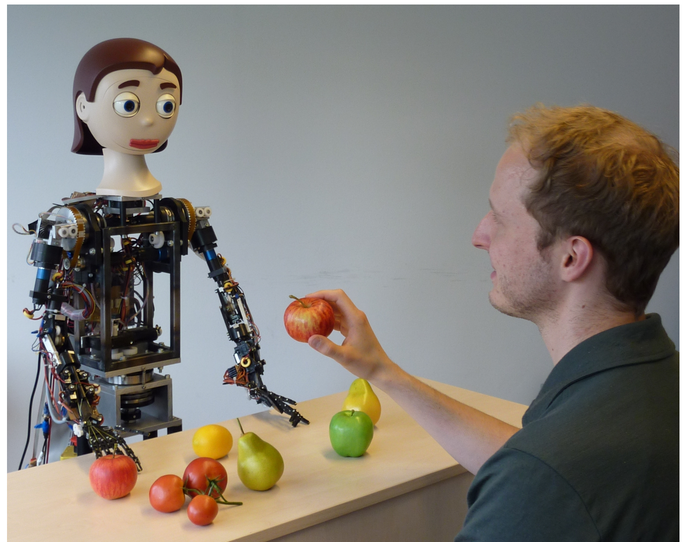

Figure 1: The Curious Robot scenario on the robot platform Flobi.

As target scenario for our case studies, we chose a simplified version of the Curious Robot, an object learning and manipulation scenario that we have presented recently (Lütkebohle et al. 2009). This is a typical robotic application as it includes problems of perception and learning as well as action oriented communication. Figure 1 shows the current setup.

As previous studies have shown, it is beneficial to allow for mixed-initiative by letting the robot ask for unknown objects and by allowing the user to initiate a teaching or a query episode at any time. Thus, whenever the robot detects an unknown object, it asks for its label (for the case studies we

have neglected the problem of reference through non-verbal gestures). Once the label is given by the user, the robot asks how to grasp the object, which the human is expected to answer by naming the grip type. Having acquired both label and grip, it autonomously grasps the object, while reporting both begin and completion or failure of the action. Grasping may also be rejected by the back-end right away, or the user may cancel the ongoing grasping action. Additionally, the user can at any time ask the robot to enumerate the objects learnt so far or how to grasp a specific object. Finally, the user ends the interaction by saying goodbye.

Although the target scenario is kept extremely simple, it presents a number of typical challenges that dialog systems in robotics have to face. First of all, the robot must *react dynamically* to its environment. Timing and order of the robot's questions can not be fixed beforehand since they depend on the robot's perception of the world. We therefore assume for our case studies that the action to select the next object comes from a back-end component, in form of an *interaction goal* which may be either *label*, *grip* or *grasp*. Second, the user's test questions require the dialog system to cope with *focus shifts* and, as they may be asked during the robot's grasping action, even with *multitasking abilities*. Finally, going on with the interaction during grasping while still enabling feedback about the on-going action and the possibility to cancel it requires some kind of *asynchronous coordination* between dialog system and back-end.

As the focus of the case studies lays on dialog modeling, the goal was not to achieve a fully fledged implementation running on a robotic platform. Therefore, speech recognition and speech synthesis were replaced by text in- and output and all perception and motor activities have been simulated. Also, we ignored subtle yet important aspects of the interaction such as nonverbal cues, social behavior or the engagement process that typically precedes the interaction.

Case study 1: Implementing the Curious Robot with Ravenclaw

The first case study investigates the Ravenclaw dialog manager which is being developed at Carnegie Mellon University (Bohus and Rudnicky 2009). A large number of speech applications have been implemented with it, spanning from a bus information system, to calendar applications, to a support application for aircraft maintenance. A well-maintained documentation including step-by-step tutorials is provided.

At the core of Ravenclaw is the *dialog task specification* which encapsulates the domain-specific aspects of the control logic and forms a hierarchical plan for the interaction and is executed by the domain-independent dialog engine at runtime. It consists of a tree of *dialog agents*, each handling a subtask of the interaction, such as greeting the user or presenting the result of a database lookup. There are two types of dialog agents: *dialog agencies* that represent tasks that are further decomposed and *fundamental dialog agents* that are terminal nodes in the tree, implementing atomic actions. The fundamental dialog agents further fall into four categories. An *Inform* agent produces an output, a *Request* agent requests information from the user, an *Expect* agent expects information from the user without explicitly requesting it, and an *Execute* agent performs back-end calls, such as database access. During interaction, the dialog engine traverses the tree in a depth-first manner, unless otherwise specified by pre- and postconditions or by error handling and repair activities. Agents from the task tree are put on top of a dialog stack in order to be executed and are eliminated when completed.

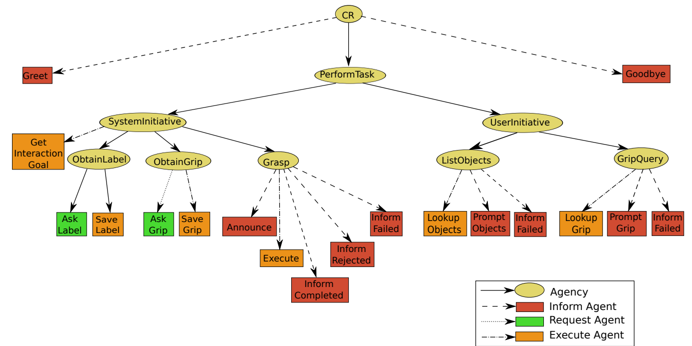

Figure 2: The Ravenclaw task specification for the Curious Robot scenario. As Ravenclaw does not support visualization, the tree was created by hand.

Figure 2 shows a possible dialog task specification for our test scenario. Its main part is the *PerformTask* agency, which is divided into two agencies handling human and robot initiative respectively. The *SystemInitiative* agency is reset after completion and executed repeatedly unless the user initiative agency is triggered or the user ends the interaction. It consists of an *Execute* agent fetching the current interaction goal from the back-end, and the agencies *ObtainLabel*, *ObtainGrip* and *Grasp*. *ObtainLabel* and *ObtainGrip* request label and grip respectively, and communicate it to the back-end where it gets stored. *Grasp* first announces grasping, then executes it and finally reports success, rejection or failure. The three agencies are not executed in succession, but alternatively, depending on conditions such as the current interaction goal (not shown in the figure). The *UserInitiative* agency can be activated by the user's test questions at any time. This is achieved by adding a trigger directive to its subagents, making *ListObjects* and *GripQuery* respectively the currently focused agent, i.e. the topmost agent on the stack. Table 1 illustrates a typical dialog example.

Technically, the dialog agents are defined as C++ macros that communicate with the back-end by exchanging user-defined frame data structures via a centralized message-passing architecture. Figure 3 illustrates the portion of the dialog task specification that defines the *Grasp* agency and its subagents *Announce*, *Execute*, *InformCompleted*, *InformRejected* and *InformFailed*. *Grasp* is only executed if the interaction goal has the value *label* (line 2), and it succeeds if the grasp action has either been rejected, completed, or has failed (lines 12-14). *Announce* prompts the speech output specified for this situation (line 17). *Execute* then calls the associated back-end function (line 21-23) and stores its result (line 24). Depending on

	Utterance	Comment
U1	Hello.	
R1	Hi, hello.	
R2	What is that?	Back-end returns interaction goal *label*
U2	This is an apple.	
R3	How should I grasp the apple?	Back-end returns interaction goal *grip*
U3	What objects do you know already?	Focus shift: *ListObject* focused agent
R4	I know 'apple'.	
R5	How should I grasp the apple?	*ObtainGrip* focused agent again
U4	How do you grasp a banana? q	Focus shift: *GripQuery* focused agent
R6	Sorry, I don't know.	
R7	How should I grasp the apple?	*ObtainGrip* focused agent again
U5	With the power grasp.	
R8	I am going to grasp the apple now.	Back-end returns interaction goal *grasp*
		Back-end blocks during grasping
R9	Sorry, I failed to grasp the apple.	
U6	Goodbye.	
R10	Goodbye.	

Table 1: An example dialog for the Curious Robot implementation with Ravenclaw, including two focus shifts for the user's test questions (U3 and U4). Since the back-end blocks during grasping (R8), an on-going grasping action can not be canceled. The failure of grasping is reported back from the back-end (R9).

whether the result is *rejected*, *completed* or *failed* (lines 28, 33, 38), the appropriate output is generated (lines 29, 34, 39).

```
1   DEFINE_AGENCY( CGrasp,
2      PRECONDITION((int)C("result.interactiongoal") == 2
3   )
4   DEFINE_SUBAGENTS(
5      SUBAGENT(Announce, CAnnounce, "")
6      SUBAGENT(Execute, CExecute, "")
7      SUBAGENT(InformCompleted, CInformCompleted, "")
8      SUBAGENT(InformRejected, CInformRejected, "")
9      SUBAGENT(InformFailed, CInformFailed, "")
10  )
11  SUCCEEDS_WHEN(
12     (SUCCEEDED(InformCompleted) ||
13      SUCCEEDED(InformRejected)  ||
14      SUCCEEDED(InformFailed)))
15
16  DEFINE_INFORM_AGENT( CAnnounce,
17     PROMPT( "inform grasping <result")
18  )
19  DEFINE_EXECUTE_AGENT( CExecute,
20     EXECUTE(
21        C("query_type") = NQ_GRASP;
22        pTrafficManager-> Call(this,
23           "backend.query <query_type >new_result");
24        C("result") = C("new_result");)
25  )
26  DEFINE_INFORM_AGENT( CInformCompleted,
27     PRECONDITION(
28        (int)C("result.taskstate") == RC_COMPLETED)
29     PROMPT( "inform grasping_completed <result")
30  )
31  DEFINE_INFORM_AGENT( CInformRejected,
32     PRECONDITION(
33        (int)C("result.taskstate") == RC_REJECTED)
34     PROMPT( "inform grasping_rejected <result")
35  )
36  DEFINE_INFORM_AGENT( CInformFailed,
37     PRECONDITION(
38        (int)C("result.taskstate") == RC_FAILED)
39     PROMPT( "inform grasping_failed <result")
40  )
```

Figure 3: Ravenclaw's dialog task specification for the *Grasp* agency and its subagents.

Most requirements of our target scenarios could be realized with Ravenclaw. When it comes to a real-world robotic scenario, a shortcoming might however be that the dialog task tree largely pre-defines the interaction flow. As suggested in our target scenario, a robot needs to react not only to the user's utterance, but also to many kinds of events that occur in its environment. With Ravenclaw, this can be achieved by controlling the navigation through the task tree with pre- and postconditions. However, for highly unstructured scenarios with many possible paths through the task tree, the dialog structure may thus become unclear, up to unstructured spaghetti code at the worst. Already our toy scenario contains a number of "jumps" in the control flow in order to react to the current interaction goal, the user's focus shifts and the back-end results.

Further, we encountered difficulties regarding the asynchronous coordination of back-end calls. While Ravenclaw does support asynchronous back-end calls, it does not provide mechanisms that support a further communication between dialog and back-end about a running back-end action. In the target scenario, grasping was therefore implemented using a blocking back-end call, which enables the robot to report success or failure when it is done. With the blocking back-end call however, the interaction can not be maintained during action execution, and also the possibility to cancel the action could not be realized.

Another issue is reusability. Even for our basic test scenario, the dialog task specification shown in figure 2 contains several agencies that have a similar structure, e.g. *ObtainLabel* and *ObtainGrip*, or *ListObjects* and *GripQuery*, and one can easily think of another agency with the same structure as the *Grasp* agency, e.g. a following or navigation task. With the *Inform*, *Expect* and *Execute* agents as the only unit of pre-modeled conversational capabilities, Ravenclaw does not account for such recurring structures, which are not specific to robotics but will occur in any domain.

A new version of the Olympus dialog architecture (in which Ravenclaw is embedded) was described briefly in (Raux and Eskenazi 2007). This new version (which is not the one we have used) features a multi-layer architecture for event-driven dialog management. It was originally designed to address the issue of reacting to conversational events in real-time so as to enable flexible turn-taking and to react on barge-ins. With the proposed architecture, also non-conversational events (e.g. perceptual events) can be handled. It therefore seems probable that some of the above difficulties could be resolved with it. In particular, with an event-based architecture, the dialog manager could react directly to a change of the current interaction goal. Also, it could react to update events of a robot action (such as *grasping begins*), while keeping the interaction going. However, it lacks an overarching structure for temporally extended actions (such as the *tasks* in the PaMini framework described in case study 4), and it lacks a generic mechanism for handling such events (such as the *task state protocol* in PaMini). This means that the event processing, i.e. keeping track of the dialog moves associated with events, is still left to the developers.

Apart from the above difficulties, Ravenclaw has proven to support many aspects of the target scenario very efficiently. For one, speech understanding integrates naturally into dialog modeling and output generation. The concepts of the semantic speech understanding grammar designed by the scenario developer are available within the dialog specification and within the output generation component. Dialog

variables need not be specified explicitly.

Further, Ravenclaw uses a generic grounding model that provides several strategies for concept grounding, such as implicit and explicit confirmation strategies, and non-understanding recovery strategies, such as repeating the original prompt, or asking the user to repeat or rephrase (Bohus and Rudnicky 2008). The grounding policies are specified in a configuration file, which is the reason why the dialog task specification in figure 2 does not contain agents for confirming and correcting label and grip.

Finally, the fact that Ravenclaw does not provide pre-modeled conversational structures can also be viewed as a benefit: the scenario developer does not have to stick to the structures provided, but has full control over the dialog flow.

Case study 2: Implementing the Curious Robot with Collagen/Disco

The second approach we looked at is the collaboration manager Collagen (for *Coll*aborative a*gen*t) (Rich and Sidner 1998). Even though it is rather a plug-in for intelligent user interfaces than a dialog system in the narrower sense, we included it in our case studies because it investigates some aspects that are very relevant for robotics, such as agents communicating about a task and coordinating their actions in order to work towards a shared goal, while accounting for physical actions as well. Unlike a real dialog system, Collagen takes rather an observational role, relying on the *collaborative interface paradigm*. In this paradigm, a software agent assists the user in operating an application program, both communicating with each other as well as with the application. They are informed about each others' actions either by a reporting communication ("I have done *x*") or by direct observation. The Collagen framework can be seen as the mediator of the communication between the agent and the user.

Various desktop applications have been developed based on Collagen, including assistants for air travel planning, email and a programmable thermostat. Our case study was however not conducted with the Collagen framework itself, but with its open-source successor Disco.

Collagen has a task model that defines for the specific application domain the typical domain goals and procedures for achieving them. The task model is a collection of goal decomposition rules, called *recipes*. Collagen tracks the user's progress with respect to the task model and automatically generates system utterances and choices for user utterances, based on the current discourse state. One component of the discourse state is the *focus stack*, representing its attentional aspects. The focus stack contains hierarchical discourse segments, each contributing to a specific *SharedPlan*. A Shared plan corresponds to the intentional aspects of the discourse and is represented as (possibly still incomplete) *plan tree*, specifying the actions to be performed, and by whom.

Figure 4 shows a collection of recipes that specify our target scenario. The upper part of the figure shows the top-level

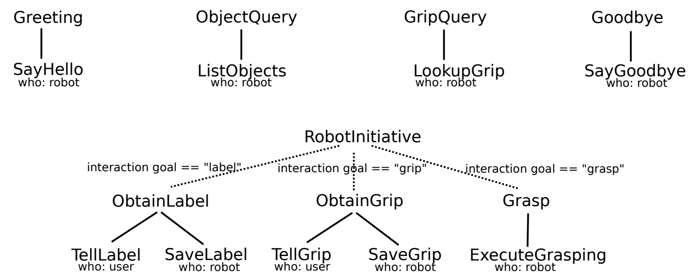

Figure 4: Collagen's recipes for the Curious Robot scenario.

goals *Greeting*, *ObjectQuery* (i.e. the user asks to enumerate the objects learnt), *GripQuery* (i.e. the user queries the appropriate grip for a specific object) and *Goodbye*, each of which can be achieved by a robot's action. For instance, the goal *Greeting* can be achieved by the robot's *SayHello* action. It may seem somewhat surprising that the mutual greeting can be achieved by the robot's *SayHello* action alone, but the user's greeting was already carried out with the user selecting the top-level goal *Greeting* (utterances U1, R1 in table 2). The top-level goal *RobotInitiative*, shown in the lower part of figure 4, covers the goals and actions concerning the robot's initiative. It is divided into the subgoals *ObtainLabel*, *ObtainGrip* and *Grasp*, each with an applicability condition over the current interaction goal. The subgoal *ObtainLabel* can be achieved with the user executing *TellLabel* and the robot executing *SaveLabel*; likewise with *ObtainGrip*. Again, it might seem surprising that the *ObtainLabel* subgoal does not imply a robot action such as *AskLabel*, but, similar as with the greeting, the robot's label query is expressed as a suggestion to the user to execute *TellLabel* (cf. table 2, utterances R2 and U3).

Listing 5 shows how the recipe for *RobotInitiative* is coded in the XML task specification language. It is decomposed into its three subtasks (lines 3, 12 and 21), which again are decomposed further (lines 5-6, 14-15 and 22). Lines 4, 13 and 22 encode the applicability conditions for the respective subtask. The value of the built-in variable *external* indicates whether it is the user or the system who is supposed to execute the subtask. For example, *TellLabel* is assigned to the user (line 7), while *SaveLabel* is assigned to the system (line 8). Further, variables can be passed from one subtask to another, for instance the label (line 9) or the grip name (line 18). A task model description may also contain JavaScript fragments that connect the model with the underlying application or device, as required for polling the current interaction goal (lines 5, 19, 33).

The dialog example shown in table 2 illustrates in detail how the dialog evolves from these recipes: the user selects the top-level goals, and the robot either performs its part of the task, if possible, or suggests an appropriate action to the user.

A fundamental difference to the implementation with Ravenclaw is that rather than the dialog flow itself, only the task needs to be specified. Based on the current discourse state and the recipes, the dialog is generated automatically out of a generic rule framework (Rich et al. 2002). Rules specify the system's next action for a particular situa-

	Configured utterance	Generated utterance	Comment
U1	Hello.	Let's achieve *Greeting*.	User selects goal *Greeting*
R1	Hello.	Ok.	Robot executes *SayHello*
U2	Let's explore the objects on the table.	Let's achieve *RobotInitiative*.	User selects goal *RobotInitiative*
			Back-end returns interaction goal *label*
R2	What is that?	Please execute *TellLabel*.	Robot asks user to perform *TellLabel*
U3	An apple.	An apple.	User asserts that *TellLabel* done
R3	Ok.	Ok.	Robot executes *SaveLabel*
U4	Let's explore the objects on the table.	Let's achieve *RobotInitiative*.	User selects goal *RobotInitiative*
			Back-end returns interaction goal *grip*
R4	How should I grasp it?	Please execute *TellGrip*.	Robot asks user to perform *TellGrip*
U5	What objects do you know already?	What objects do you know already?	Focus shift: User selects goal *ObjectQuery*
R5	Ok.	Ok.	Robot executes *ListObjects*
R6	How should I grasp it?	Please execute *TellGrip*.	Back to *TellGrip*
U6	With the power grasp.	With the power grasp.	User asserts that *TellGrip* done
R7	Ok.	Ok.	Robot executes *SaveGrip*
U7	Let's explore the objects on the table.	Let's achieve *RobotInitiative*.	User selects goal *RobotInitiative*
			Back-end returns interaction goal *grasp*
			Robot executes *Grasp*
R8	Ok.	Ok.	*Grasp* failed
U8	Goodbye.	Let's achieve *Goodbye*.	User selects goal *Goodbye*
R9	Goodbye.	Ok.	Robot executes *SayGoodbye*

Table 2: Example dialog for the Curious Robot implementation with Collagen/Disco.

```
1   <task id="RobotInitiative">
2
3      <subtasks id="ObtainLabel">
4          <applicable> interactionGoal() == "label" </applicable>
5          <step name="TellLabel" task="TellLabel"/>
6          <step name="SaveLabel" task="SaveLabel"/>
7          <binding slot="$TellLabel.external" value="true"/>
8          <binding slot="$SaveLabel.external" value="false"/>
9          <binding slot="$SaveLabel.label" value="$TellLabel.label"/>
10     </subtasks>
11
12     <subtasks id="ObtainGrip">
13         <applicable> interactionGoal() == "grip" </applicable>
14         <step name="TellGrip" task="TellGrip"/>
15         <step name="SaveGrip" task="SaveGrip"/>
16         <binding slot="$TellGrip.external" value="true"/>
17         <binding slot="$SaveGrip.external" value="false"/>
18         <binding slot="$SaveGrip.grip" value="$askGrip.grip"/>
19     </subtasks>
20
21     <subtasks id="Grasp">
22         <applicable> interactionGoal() == "grasp" </applicable>
23         <step name="ExecGrasping" task="ExecGrasping"/>
24         <binding slot="$ExecGrasping.external" value="false"/>
25         <binding slot="$success" value="$ExecGrasping.success"/>
26     </subtasks>
27
28  </task>
```

Figure 5: Collagen's recipes for the Robot Initiative goal.

tion. For instance, the *Execute* rule specifies that a primitive task that is assigned to the agent should be executed directly, whereas the *AskWho* rule says that for a task whose executor is not determined, the system should return an utterance of the form ”Who should perform *goal*?”. Collagen provides a collection of default rules, and further rules can be plugged in to implement a different collaboration style.

The generated output can be customized as to how tasks can be referred to and how their execution is confirmed. Table 2 contrasts the customized version of the output with the automatically generated version, e.g. ”Hello” versus ”Let's achieve *Greeting*” in utterance U1. Additionally, the rules generate not only the system's next action but present also the agenda for the user, i.e. choices for the user to say, or rather to type. For example, choices generated after the robot's label query include rejecting the proposed action (”I'm not going to anwer your question”), abandoning the top-level goal (”Let's not explore the objects on the table”) and focus shifts (”What objects do you know already?”, ”How do you grasp a banana?”).

Although our target scenario is not at all the type of scenario Collagen was intended for originally, many requirements of the target scenario could be realized with it. Its model of collaborative discourse, wherein two interaction partners collaborate on a task by proposing and performing actions, supports very well the focus shifts that were stipulated in the specification. In contrast, the robot's task initiative that generates its query for label and grip could not be implemented using the default agent that comes with the framework since it does not suggest top-level goals on its own. It should however be easily possible to adapt the default implementation such that it is able to propose *ObtainLabel*, *ObtainGrip* and *Grasp* autonomously. For the case study, we worked around this problem by introducing the top-level goal *RobotInitiative*, which the user is to select explicitly (”Let's explore the objects on the table.”), whereupon the robot chooses between *ObtainLabel*, *ObtainGrip* and *Grasp*, depending on the current interaction goal.

Another problem we encountered affects the communication of back-end results, such as the success of grasping or the robot's enumeration of the objects learnt. Collagen does not support variable system utterances, e.g. by template-based output generation. This is the reason why the robot simply answers *Ok* when the user asks to enumerate the known objects (cf. R5 in table2), or why the robot does not communicate that grasping has failed (cf. R8 in table 2). Admittedly, Collagen does not claim to be a complete natural-language processing system, and within the collaborative interface-agent paradigm it would probably be the underlying application that is responsible for representing the application-specific results to the user.

The automatic generation of system utterances is a very

```
1  infostate(record([is:record([
2     utterance:atomic,
3     interpretation:atomic,
4     task_event:atomic,
5     interaction_goal:atomic,
6     listening:atomic,
7     awaiting_event:atomic])])).
8
9  urule(getInteractionGoal,
10    [eq(is:interaction_goal,'')],
11    [solve(getInteractionGoal(X),
12     [assign(is:interaction_goal,X)]),]).
13
14 urule(waitForUtterance,
15    [eq(is:listening,yes)],
16    [solve(recognize(X, Y),
17     [assign(is:utterance, X),
18      assign(is:interpretation, Y),
19      assign(is:listening, no)])]).
20
21 urule(processLabel,
22    [eq(is:interpretation,label),
23     eq(is:interaction_goal, label)],
24    [solve(store(is:utterance)),
25     solve(say(is:utterance Okay)),
26     assign(is:interaction_goal, ''),
27     assign(is:utterance, ''),
28     assign(is:interpretation, '')]).
29
30 urule(LabelQuery,
31    [eq(is:interaction_goal,label)],
32    [solve(say('What is that'),
33     [assign(is:listening,yes)])]).
```

Figure 6: Dipper's information state definition and update rules for the robot's label query.

powerful technique. However, while the wording of the generated utterances can be configured, the developer can not control *when* utterances are generated. This is the reason why the begin of grasping can not be announced (cf. R8 in table 2). Also, generating utterances automatically leads to asymmetry in the task model: while some of the user utterances are explicitly represented as subtasks (e.g. *TellLabel* and *TellGrip*), the system utterances are not present in the task model.

The most serious shortcoming pertains to error handling. The task model provides a built-in *success* variable, indicating the success of a subtask. It is used to control replanning. However, a binary value might not always provide sufficient information. Some applications might want to discriminate between a failure and a rejection of the subtask, or between different error causes. For instance, if a plan fails because the underlying application is otherwise busy, it might be reasonable to re-execute the plan later, whereas retrying might be pointless if the requested functionality is unavailable in general.

Finally, just as the Ravenclaw framework, Collagen does not provide mechanisms for asynchronous coordination of task execution. Thus, neither the continuation of the interaction during grasping could be realized, nor could the grasping action be canceled.

Case study 3: Implementing the Curious Robot with Dipper

In the third case study, we explored the Information State (IS) approach to dialog modeling (Traum and Larsson 2003) whose key idea is that the dialog is driven by the relevant aspects of information (the *information state*) and how they are updated by applying *update rules*, following a certain *update strategy*. The term *information state* is intentionally kept very abstract. One may choose to model the external aspects of the dialog, such as variables to assign, or rather the internal state of the agents, such as goals, intentions, beliefs and obligations, in order to realize a plan-based dialog management. The Prolog-based TrindiKit is known as the original implementation of the IS approach (Traum and Larsson 2003). Others followed, based on different programming languages. For our case study, we chose the stripped-down re-implementation Dipper (Bos et al. 2003).

Dipper is set on top of the Open Agent Architecture (OAA), a C++ framework for integrating different software agents in a distributed system (Martin, Cheyer, and Moran 1999). OAA agents provide services that other agents may request by submitting a high-level Interagent Communication Language (ICL) expression (a *solvable*, which can be viewed as a service request) to the *facilitator* agent that knows about all agents and mediates the interaction between them. In addition to the *facilitator* and the *Dipper* agent, the implementation of the target scenario includes a *SpeechRecognitionAgent* and a *TTSAgent* for (simulated) speech in- and output, a *MotorServer* agent that simulates grasping, an *ObjectDatabase* that stores object labels and the associated grip, and an *ActionSelection* agent that selects the current interaction goal.

The upper part of listing 6 (lines 1-7) shows the information state for the Curious Robot scenario, which is designed such that it models the most obvious information, namely the current interaction goal (line 5), the current user utterance and its interpretation (line 2-3), and incoming events from the back-end task (line 4). Further, it contains control flags that determine whether the system is ready to receive speech input (line 6) or task events (line 7).

The lower part of listing 6 (lines 9-33) shows the update rules that are necessary to realize the robot's label query. Update rules are written in the Prolog-like Dipper update language, specified by the triple $\langle$*name, conditions, effects*$\rangle$, with *name* a rule identifier, *conditions* a set of tests on the current information state, and *effects* an ordered set of operations on the information state. The first rule, *getInteractionGoal*, deals with the situation when no interaction goal is set (line 10). In that case, an OAA solvable is sent that polls the interaction goal (line 11) and updates the information state with the result (line 12). The second rule, *waitForUtterance*, is applicable if the *listening* flag is set (line 15). It posts a solvable for the *SpeechRecognitionAgent* (line 16), integrates the result into the information state (lines 17-18) and resets the flag (line 19). The *processLabel* rule applies if the user has given an object label (line 22-23). It posts solvables for acknowledging and storing the label (lines 24-25) and resets the information state (lines 26-18). The last rule, *LabelQuery*, posts a solvable that will trigger the label query and set the flag for receiving speech input (lines 32-33), if the current interaction goal is *label* (line 31).

When implementing the target scenario, we found the idea of a central information state that determines the next steps of the interaction to be very intuitive. Also, the division of responsibilities between distributed agents enables a modular approach that roughly resembles the distributed event-based architecture of the original system.

However, we encountered problems with respect to the update rules and the update strategy. While TrindiKit leaves

it to the developer to implement the (possibly highly complex) update strategy, Dipper provides a built-in update strategy that simply selects the first rule that matches, applies its effects to the information state, and starts over with checking the first rule again. This means that rules are executed on a first-come, first-served principle, where the order of the rules matters, resulting in a brittle system behavior. In our case study, this is the reason why e.g. the *processLabel* rule is defined before the *LabelQuery* rule. If it was the other way round, the system would loop over the *LabelQuery* and never execute *ProcessLabel*. Of course, we could overcome the problem by introducing additional control flags, which would make the information state unnecessarily complex. As a result of this update strategy, some requirements of the target scenario could not be realized.

Focus shifts could only partly be implemented. The problem was not to define the appropriate update rules (e.g. *processListObjects*), but rather that user utterances are processed only at specific points in time, that is, only if the *listening* flag (which we have adopted from the example in (Bos et al. 2003)) is set. Thus, a focus shift may be initiated only when the robot expects the user to speak, e.g. after having asked the label query. If the rule for speech recognition was applicable at any time, it might conflict with other rules.

Also, asynchronous coordination, which the OAA framework actually supports well, could only partly be realized, due to the first-come, first-served update strategy that enables speech input only at certain points. Thus, the robot's feedback on the grasping action could be realized by explicitly waiting for respective task events by virtue of the *waitForTaskEvent* rule, whereas the possibility to cancel an ongoing grasping action could not be implemented because the *waitForUtterance* rule would have conflicted with the *waitForTaskEvent* rule.

Another issue is that the update rules handle both organizational tasks (such as polling different input sources or producing output) and dialog management tasks. A clear separation of concerns could make the dialog strategy more obvious and prevent the information state from being overloaded with control flags.

In a real-world application, testability and maintainability might become issues. As rule systems get more complex, their behavior can become very hard to predict. Already in our simplified Curious Robot implementation, which required about 15 rules, it was not easy to identify the actual dialog flow.

Case study 4: Implementing the Curious Robot with PaMini

Finally, we re-implemented the target scenario with the PaMini approach that we have suggested recently (Peltason and Wrede 2010b), (Peltason and Wrede 2010a). In contrast to the above dialog frameworks, PaMini targets specifically human-robot interaction.

One key idea of the approach is the concept of *tasks* that can be performed by components. Tasks are described by an execution state and a task specification that contains the information required for execution. A *task state protocol* specifies task states relevant for coordination and possible transitions between them. Typically, a task gets *initiated*, *accepted*, may be *canceled* or *updated*, may deliver intermediate *results* and finally is *completed*. Alternatively, it may be *rejected* by the handling component or execution may *fail*. Task updates cause event notifications which are delivered to PaMini (and to other participating components). From dialog system perspective, the task state protocol establishes a fine-grained – yet abstract – interface to the robotic subsystem. PaMini is backed by the integration toolkit XCF (Wrede et al. 2004), but could in principle be used with any middleware that supports the task state protocol.

Internally, PaMini relies on generic *interaction patterns* that model recurring conversational structures, such as making a suggestion or negotiating information. They are defined at an abstract level, but can be tailored with an application-specific configuration. Interaction patterns are visualized as a finite-state transducer, taking as input either human dialog acts or the above task events, and producing robot dialog acts as output. In the states, actions such as task or variable updates may be executed. By combining the task states with robot dialog acts, the conversation level is related with the domain level. At run-time, patterns can be interleaved to achieve a more flexible interaction style.

The interaction patterns have been extracted from different human-robot interaction scenarios, most notably a home-tour scenario in which a mobile robot acquires information about its environment, and the (original) Curious Robot scenario. Since then, PaMini has been used in many further scenarios, ranging from a virtual visitor guide to a robot assistant that has been sent to the RoboCup@Home 2011 competition. Technically, the patterns are implemented as statecharts (Harel 1987), using the Apache Commons SCXML engine (Apache Commons 2007), and wrapped in a Java API.

To realize the target scenario, we have set up a distributed system with (simulated) components that communicate via the task state protocol. The system includes components for speech in- and output, an action selection component, and a motor server. This breakdown is similar to the one in the Dipper-based implementation.

When selecting the interaction patterns to use, we found that in some cases more than one pattern provided by PaMini was appropriate. For instance, there are a few patterns modeling a robot information request, differing in the confirmation strategy (implicit or explicit). Also, the system-initiated action may be cancelable or not, and the robot may ask for permission before action execution or not. For our target scenario, we chose an explicit confirmation strategy and non-acknowledged yet cancelable action execution. Altogether, we used seven patterns, one each for greeting, parting, the robot's label and grip query, the user's test questions and the robot's self-initiated grasping.

The upper part of figure 7 shows the *Robot Self-Initiated Action* pattern, required for the grasping action. First, the robot announces its intended action. Next, PaMini initiates the associated system task. As the handling system component accepts the task, the robot asserts action execution.

Once the task is completed, the robot acknowledges. Additionally, the pattern handles the various possible faults. Listing 8 shows an excerpt of the associated dialog act configuration. It determines conditions for the human dialog acts and how exactly the robot dialog acts should be expressed, both being possibly multimodal. The dialog act *R.acknowledge* in the state *asserted*, for instance, is specified as the utterance *I finished grasping* (lines 1-5). Similarly, in order to be interpreted as *H.cancel*, the XML representation of the user utterance has to match the XPath expression */utterance/cancel* (lines 13-16).

Apart from the dialog acts, the developer has to configure the task communication (i.e. the task specification for tasks initiated by the dialog system, and possible task state updates), as well as the definition of variables (used for parameterizing the robot's dialog acts and within the task specification). While the dialog act configuration is written in the domain-specific XML configuration language as depicted in listing 8, the latter two are specified by extending Java base classes. Since not each pattern involves task communication or the use of variables, only the dialog act configuration is obligatory.

The definition of variables and the task communication often goes hand in hand. For instance, the robot's label query is modeled using the *Correctable Information Request* pattern, shown in the lower part of figure 7. As soon as the human answers the robot's information request (*R.question* and *H.answer*, respectively), the object label is extracted from the user utterance and assigned to an appropriate variable (cf. *update-variable-context* in state *await_confirmation*), which is then used to parameterize the robot's confirmation request *R.askForConfirmation* (e.g. 'Apple. Is that correct?'), and to augment the task specification with the label so as to transfer it to the responsible system component.

As the Curious Robot has been one of the development scenarios for PaMini, it is not surprising that all of the stipulated requirements could be met. With the task state protocol, state updates of temporally extended back-end calls such as grasping are delivered by event notification, enabling PaMini to give feedback on the on-going action, as illustrated in table 3 (R8, R10). Conversely, PaMini can update or cancel tasks on-line (U7-R10). By admitting interleaving interaction patterns, the interaction can be maintained during task execution. During the robot's grasping action, for instance, the user initiates a focus shift by asking a question (U6-R9), which is modeled by interleaving a *Robot Self-Initiated Action* pattern with a *Human Information Request*.

Perhaps the most striking difference to the other dialog frameworks affects discourse planning. While *local* discourse planning is determined by the interaction patterns, *global* discourse planning – i.e., how the patterns are combined – is not done within the dialog framework, but is decided by the back-end (or by what the user says, of course). This enables the dialog system to respond to the dynamic environment in a flexible and reactive way.

While the other frameworks discussed provide generic strategies for grounding or collaboration, PaMini goes one step further in this respect by providing pre-modeled "building blocks of interaction", intended to encapsulate the subtleties of dialog management and domain integration. Both a usability test (Peltason and Wrede 2010a) and our first experiences with the framework support that interaction patterns enable developers to rapidly implement new interaction scenarios. Furthermore, as the interaction patterns are kept self-contained, new features can be added without breaking existing functionality (of which we are running the risk e.g. with Dipper's update rules). This significantly eases incremental system development.

```
1   <robotDialogAct
2      state="asserted"
3      type="R.acknowledge">
4         <verbalization
5            text="I finished grasping."/>
6   </robotDialogAct>
7
8   <robotDialogAct
9      state="asserted"
10     type="R.apologize">
11        <verbalization
12           text="Sorry, I failed to grasp the %OBJECT%."/>
13  </robotDialogAct>
14
15  <humanDialogAct
16     state="asserted"
17     type="H.cancel"
18     xpath="/utterance/cancel"/>
```

Figure 8: An excerpt from PaMini's dialog act configuration for the robot's grasping action.

As PaMini has been developed with robotics in mind, it might exhibit a couple of deficiencies when applied to non-robotics scenarios. First, the definition of variables is not as straightforward as e.g. with Ravenclaw, where variables are derived directly from the semantic speech recognition grammar. PaMini, in contrast, leaves variable handling to the developer. Also, as PaMini does not maintain an explicit representation of the information that needs to be gathered during the dialog, overanswering is not supported. Relying on tasks as the fundamental concept that drives the interaction, PaMini could be referred to as an *action-oriented*, rather than as *information-oriented* approach.

Also, as PaMini outsources much of the responsibility for discourse planning to the back-end, little support is provided for interactions whose structure is determined by the current dialog state (or the current *information state*) rather than by an "intelligent back-end". This might be a hassle in information negotiating scenarios, in which the back-end typically plays a more passive role.

In general, the interaction patterns, how useful they may be, definitely impose restrictions on the developers. Though patterns for many purpose do exist, the framework is not designed such that new patterns can be implemented by non-experts easily.

Discussion

Our case study was performed with the goal to identify pitfalls and potential remedies as well as potential future challenges for dialog modeling on robots. As the case studies have shown, none of the investigated frameworks overcomes all problems in one solution. However, the summary of the results as depicted in Table 4 allows to draw

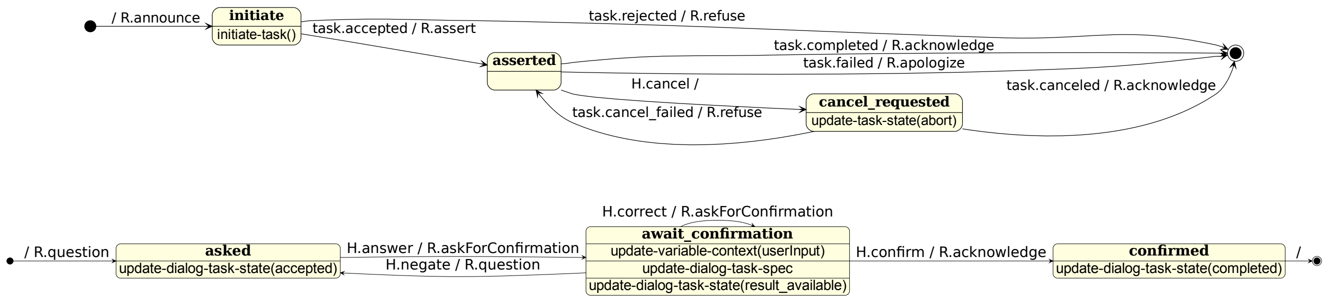

Figure 7: Two example interaction patterns. The upper part shows the *Robot Self-Initiated Cancelable Action* pattern, which is used to model the robot's grasping action. The lower part shows the *Robot Information Request With Explicit Confirmation* pattern, which is used to model the robot's label and grip queries.

	Utterance	Comment	Active Interaction Pattern
U1	Hello.		Human Interaction Opening
R1	Hi, hello.		Human Interaction Opening
R2	What is that?	Back-end publishes interaction goal *label*	Robot Information Request with Explicit Confirmation
U2	This is a lemon.		Robot Information Request with Explicit Confirmation
R3	Melon. Is that correct?		Robot Information Request with Explicit Confirmation
U3	No, it's a lemon!		Robot Information Request with Explicit Confirmation
R4	Lemon. Is that correct?		Robot Information Request with Explicit Confirmation
U4	Yes, this is correct.		Robot Information Request with Explicit Confirmation
R5	How should I grasp the lemon?	Back-end publishes interaction goal *grip*	Robot Simple Information Request
U5	With the power grasp.		Robot Simple Information Request
R6	Alright.		Robot Simple Information Request
R7	I am going to grasp the lemon.	Back-end publishes interaction goal *grasp*	Robot Self-Initiated Cancelable Action
		Grasp task *initiated* by PaMini	Robot Self-Initiated Cancelable Action
R8	I start grasping now.	Grasp task *accepted* by back-end	Robot Self-Initiated Cancelable Action
U6	How do you grasp a banana?	Focus shift and multi-tasking	Human Information Request
R9	Sorry, I don't know.		Human Information Request
U7	Stop!	*Cancel requested* by PaMini	Robot Self-Initiated Cancelable Action
R10	Ok, I stop.	Grasp task *canceled* by back-end	Robot Self-Initiated Cancelable Action
U8	Goodbye.		Human Interaction Closing
R11	Goodbye.		Human Interaction Closing

Table 3: An example dialog for the Curious Robot implementation with PaMini. Not surprisingly, all requirements could be realized, including focus shifts, multitasking, and cancellation of an on-going grasping action.

some interesting conclusions. On the one hand there are challenges that have been solved by most of the four dialog frameworks such as focus shifts, grounding and separation between dialog and task structure. On the other hand, the question how to model the interaction with the back-end tends to be solved individually by each framework. Binary success variables, as used in the reasoning-based Collagen/Disco approach seem to be somewhat underspecified for a satisfying information behavior of the robot. The user-defined result frame allows for more freedom but also imposes much knowledge and work on the developer. From this perspective PaMini's task state protocol appears to be a good compromise between both, allowing the developer an easy and standardized yet flexible interaction with the back-end. Depending on the solution for the interplay with the back-end, the discourse planning – which is an important factor for the user experience – is affected in different ways: frameworks that are more back-end driven tend to allow for a less restricted dialog structure.

In order to keep our case studies simple, we have limited the target scenario to verbal interaction. Nevertheless, nonverbal behaviors and multimodality are crucial aspects in situated dialog. Except for Collagen/Disco, which relies on text in- and output, multimodality could have been realized with all of the discussed dialog managers, as they operate at the semantic level below which the in- and output sources may be exchanged. The new version of Ravenclaw supports multimodal in- and output by providing agents for modality integration and for the production of multimodal output (Raux and Eskenazi 2007). Both Dipper and PaMini rely on a distributed architecture with arbitrary sources for in- and output. PaMini, for instance, provides a collection of available output modalities such as pointing gestures or mimics (depending on the robot platform), that can be combined. However, neither Dipper nor PaMini handles the issues of input fusion and output synchronization.

Moreover, human-robot interaction demands more than classical 1:1 interactions. Often, the robot will be situated in environments where multiple possible interaction partners are present, or a robot might even have to collaborate with other robots. Thus, the capability of multi-party interaction

is another crucial requirement. PaMini has recently been extended so as to be able to manage multiple interactions (with multiple participants each), and a multi-party engagement model (Bohus and Horvitz 2009) has been integrated (Klotz et al. 2011). Ravenclaw has provisions for the opposite case, in which multiple robots collaborate, forming a team (Dias et al. 2006).

Overall, the results and insights from our case studies indicate that the focus of future research will lie on the semantic processes, such as reasoning or back-end commnication. This relates to the questions that we have not targeted in our present investigation, such as how to couple reasoning with dialog structure. This entails – among others – how to make reasoning processes transparent to the user without overloading answers with technical details, but also how to interactively incorporate user input into the reasoning process. Semantic processes also relate to back-end communication which grounds the communication in physical, real-world actions and thus opens up new possibilities with respect to core linguistic questions of semantic respresentation, reference resolution or non-verbal communication. From this perspective we can expect human-robot interaction to have a boosting effect on dialog modeling research.

Acknowledgments

This work has partially been supported by the Center of Excellence for Cognitive Interaction Technology (CITEC), funded by the German Research Foundation (DFG), and the German Aerospace Center with financial support of the Federal Ministry of Economics and Technology due to a resolution of the German Bundestag by the support code 50 RA 1023. Further partial support comes from the FP7 EU project HUMAVIPS, funded by the European Comission under the Theme Cognitive Systems and Robotics, Grant agreement no. 247525.

References

Apache Commons. 2007. Commons scxml. [Online: http://commons.apache.org/scxml; accessed 27-Jul-2011].

Bohus, D., and Horvitz, E. 2009. Models for multiparty engagement in open-world dialog. In *SIGDIAL 2009 Conference*, 225–234. Association for Computational Linguistics.

Bohus, D., and Rudnicky, A. I. 2008. Sorry, I Didn't Catch That! 123–154.

Bohus, D., and Rudnicky, A. I. 2009. The RavenClaw dialog management framework: Architecture and systems. *Computer Speech & Language* 23(3):332–361.

Bos, J.; Klein, E.; Lemon, O.; and Oka, T. 2003. DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *In 4th SIGdial Workshop on Discourse and Dialogue*, 115–124.

Dias, M. B.; Harris, T. K.; Browning, B.; Jones; Argall, B.; Veloso, M.; Stentz, A.; and Rudnicky, A. 2006. Dynamically Formed Human-Robot Teams Performing Coordinated Tasks. In *AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone*.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8:231–274.

Klotz, D.; Wienke, J.; Peltason, J.; Wrede, B.; Wrede, S.; Khalidov, V.; and Odobez, J.-M. 2011. Engagement-based multi-party dialog with a humanoid robot. In *SIGDIAL 2011 Conference*, 341–343. Association for Computational Linguistics.

Lütkebohle, I.; Peltason, J.; Schillingmann, L.; Wrede, B.; Wachsmuth, S.; Elbrechter, C.; and Haschke, R. 2009. The curious robot - structuring interactive robot learning. In *IEEE International Conference on Robotics and Automation*, 4156 –4162.

Martin, D. L.; Cheyer, A. J.; and Moran, D. B. 1999. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 13(1-2):91–128.

Peltason, J., and Wrede, B. 2010a. Modeling human-robot interaction based on generic interaction patterns. In *AAAI Fall Symposium: Dialog with Robots*.

Peltason, J., and Wrede, B. 2010b. Pamini: a framework for assembling mixed-initiative human-robot interaction from generic interaction patterns. In *SIGDIAL 2010 Conference*, SIGDIAL '10, 229–232.

Raux, A., and Eskenazi, M. 2007. A multi-layer architecture for semi-synchronous event-driven dialogue management. In *IEEE Workshop on Automatic Speech Recognition & Understanding*, 514–519.

Rich, C., and Sidner, C. L. 1998. COLLAGEN: A Collaboration Manager for Software Interface Agents. *User Modeling and User-Adapted Interaction* 8:315–350.

Rich, C.; Lesh, N.; Garland, A.; and Rickel, J. 2002. A plug-in architecture for generating collaborative agent responses. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, 782–789.

Traum, D., and Larsson, S. 2003. *The Information State Approach to Dialogue Management*. Kluwer Academic Publishers. 325–353.

Wrede, S.; Hanheide, M.; Bauckhage, C.; and Sagerer, G. 2004. An active memory as a model for information fusion. In *7th International Conference on Information Fusion*.

Julia Peltason is a research assistant in the Applied Informatics Group at Bielefeld University, Germany, where she is working towards her Ph.D. in the field of human-robot interaction. Her research interests include dialog system engineering, with a focus on the specific demands of advanced human-robot interaction, such as mixed-initiative interaction, situated dialog and tight integration of action execution into interaction. Central to her work is also the question of how to design dialog systems in a way that makes them easy configurable and easy to apply even for non-dialog people. Moreover, she is interested in interaction design and has contributed to the implementation of several robot systems. Julia received her Diploma Degree in Informatics at the University of Freiburg, Germany, in 2006.

Britta Wrede is currently head of the Applied Informatics Group and responsible investigator in the Center of Excellence for Cognitive Interaction Technology (Citec) and the Research Institute for Cognition and Robotics (CoR-Lab) at Bielefeld University. Her research interests focus on modeling and analysing human-robot interaction including dialog modeling, speech recognition and emotional aspects of HRI. This also entails the question how to endow robots with percpetual and interactive capabilities to exploit ostensive signals from a human tutor in order to incrementally build an understanding of a demonstrated action. She is

	Ravenclaw	***Collagen/Disco***	***Dipper***	***PaMini***
Type of approach	Descriptive	Plan-based	Plan-based	Descriptive
Dialog specification	C++ Macros	XML	Prolog-like Dipper update language	Java and XML
Back-end specification	Arbitrary components	JavaScript	Arbitrary components	Arbitrary components
Discourse planning	Task tree	Recipes	Information state update rules	Locally: Interaction Patterns, globally: Back-end
Communication with Back-end	User-defined result frame	Optional binary *success* variable	Interagent Communication Language	Task State Protocol
Relationship between dialog and task structure	Separated	Dialog structure emerges from task structure	Separated	Separated
Pre-modeled conversational skills	Grounding and repair	Collaborative plan execution	No	Patterns for various situations
Visualization	No	No	No	Yes
Grounding model	Explicit	None	Explicit	Implicit
Plan recognition, planning	No	Yes	Yes	No
Focus shifts	Yes	Yes	Partial	Yes
Asynchronous Coordination	No (latest version: Yes)	No	Yes (polling-based)	Yes (event-based)
Multimodality	Yes	No	Yes	Yes
Multi-party interaction	Yes, n systems:1 user	No	No	Yes, 1 system:n users

Table 4: Comparison of dialog modeling approaches.

responsible investigator in several EU and national robotics projects.