

Flexible Speech Act Based Dialogue Management

Eli Hagen and Fred Popowich

School of Computing Science

Simon Fraser University

Canada V5A 1S6

{hagen,popowich}@cs.sfu.ca

Abstract

We present an application independent dialogue engine that reasons on application dependent knowledge sources to calculate predictions about how a dialogue might continue. Predictions are language independent and are translated into language dependent structures for recognition and synthesis. Further, we discuss how the predictions account for different kinds of dialogue, *e.g.*, question-answer or mixed initiative.

1 Introduction

The computerized spoken information systems (or Spoken Dialogue System—SDS) that we will consider in this paper are systems where a computer acts as the operator of some service and interacts with a user in natural language, *e.g.*, switch board, directory assistance, or ticket service. Before an SDS can provide its information, it needs to acquire data from the user, *e.g.*, customer name and number, birth date, service location, or service date. We call these parameter values. In an SDS they are acquired orally and speech recognition is used to decode the speech signal into words.

A dialogue manager facilitates the negotiation of parameter values between a user and an SDS. We emphasize keeping our dialogue manager *application and language independent*, thus we factored out the independent information into two components. A *dialogue engine* calculates predictions for how to continue a dialogue from dependent knowledge sources (*e.g.*, dialogue grammar and history, application description). A *pragmatic interpreter* maps syntactic/semantic interpretation results onto predictions.

Our predictions are called *dialogue primitives*; GEN-primitives predict system utterances and REC-primitives predict user utterances. They are language independent and on both the recognition and the generation side, other modules translate them into language dependent structures. In this paper, we will discuss the kinds of primitives our dialogue manager calculates and how

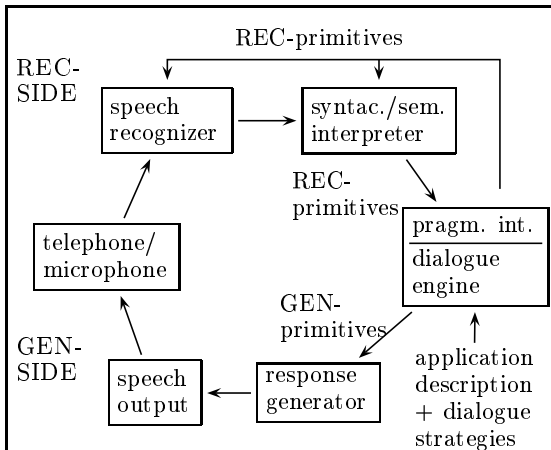


Figure 1: System architecture of our SDS. The arcs indicate information flow.

they account for different kinds of dialogue, *e.g.*, question-answer or mixed initiative.

2 Background

First, we discuss our system architecture and data flow between modules. Second, we present the application description of a movie service, which we will use for the examples in later sections. Third, we present some of our current primitives, and finally, we describe the dialogue engine and how it uses the application description and other sources to calculate dialogue primitives.

2.1 System Architecture

Our system architecture is presented in Figure 1. The dialogue manager takes an application description (Section 2.2) and a set of dialogue strategies (Sections 3 and 4) as input—both provided by the service designer. The application description describes the parameters needed by the service and is necessarily application dependent. The dialogue strategies contain directions for how the dialogue shall proceed in certain situations. For example, whether to ask for confirmation or spelling of a badly recognized parameter value or whether

to generate system or user directed dialogue.

The output of our dialogue manager is a bag of abstract, language independent primitives. On the generation side they encode the next system utterance and a response generator translates the GEN-primitives into text, which is then synthesized. On the recognition side, the REC-primitives represent the dialogue manager's predictions about the next user utterance. REC-primitives are translated into (recognition) contexts and grammars for speech recognition and they may activate sub-components of a synsem grammar. After speech recognition has taken place, the dialogue engine must be told which predictions came true, thus the pragmatic interpreter maps the output of synsem interpreter onto a sub-bag of REC-primitives, which is then returned to the dialogue engine for further processing (Section 2.4).

2.2 Application Description

The application description (AD) specifies the tasks that a service can solve and the parameter values needed to solve them. The AD for a movie service is presented in Figure 2. Our representation is an extended version of and-or trees¹ and in Figure 2, the \cup -shaped symbols represent *and*-relations, while the \vee -shaped symbols represent *or*-relations. Thus, this movie service can perform three tasks: selling tickets *or* providing movie *or* theatre information. If the user wants to buy tickets, the system needs to acquire six parameter values, *e.g.*, the show time, the date, *and* the name of the film. Date and show time can be acquired in several ways. For example, a date can be a simple date (*e.g.*, "November 17th") *or* a combination of day of the week *and* week (*e.g.*, "Wednesday this week.>").

The nodes keep state information. *Open* nodes have not yet been negotiated, *topic* nodes are being negotiated, and *closed* nodes have been negotiated. The currently active task has status *active*. Parameters can be retrieved through the functions *activeTask(AD)*, *openParams(AD)*, *closedParams(AD)*, and *topicParams(AD)*. *Status(p)* returns the status of parameter p . *tasks(AD)* and *params(AD)* return the task and parameter nodes.

Similar hierarchical domain descriptions have been suggested in (Young et al., 1990) for a naval domain and in (Caminero-Gil et al., 1996) for an e-mail assistance domain. A tree-like organization of the domain is sufficient for the information retrieval domains, which we are currently considering. We expect, however, that in future work we

¹Extensions include has-a relations.

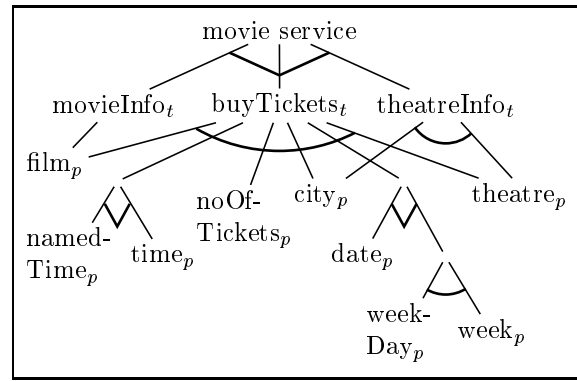


Figure 2: A description of a movie service. Notation: \cup and \vee represent *and/or*-relations. Subscripts $_t$ and $_p$ denote tasks and parameters.

will need to switch to a semantic network structure or since our future research includes automatic generation of system utterances from our dialogue primitives, we hope to be able to utilize the ontology and domain organization work, which has proven so useful for text generation (Bateman et al., 1994; Bateman et al., 1995), for both dialogue management and text generation.

2.3 Dialogue Primitives

Following the procedure outlined in Section 2.4, the dialogue manager calculates a bag of primitives for each turn and speaker. Our current collection is motivated through our experience with several domains, *e.g.*, movie service, horoscope service, and directory assistance. The collection is not exhaustive and we will add primitives as wider dialogue coverage is required.

Notation: A primitive is written *primName(p=v,n)*, where *primName* is its name; $p \in params(AD) \cup \{aTask\}$; *aTask* is a special parameter whose values $\in tasks(AD)$; v is the value of p ; and n is an integer denoting the number of times a primitive has been uttered. If v is uninstantiated, it is left out for readability. Unless otherwise stated, $p \in params(AD)$.

2.3.1 GEN-Primitives

Our current GEN-primitives:

salutation(p=v): system opens or closes the interaction. $p \in \{\text{hello, goodbye}\}$. $v \in \{\text{morning, day, evening}\}$.

requestValue(p): system requests a value for the parameter p . $p \in params(AD) \cup \{aTask\}$.

requestValue(p=v): system asks whether the value v of parameter p is correct. If this form is used, the system has a list of alternative values for p , and

v is not a recognition result (*e.g.*, Frankfurt am Main or Frankfurt an der Oder where Frankfurt is the recognition result.)

requestValue(aTask=v), $v \in \text{tasks}(AD) \cup \{\text{repeatPreServiceTask}, \text{useService}, \text{repeatService}\}$: system requests a value for $aTask$. If $v \in \{\text{repeatPreServiceTask}, \text{useService}, \text{repeatService}\}$, the system requests whether the user wants the pre-service task repeated, the service started (first task after pre-service task), or a new task started.

requestConfirm(p=v): system asks whether the value v of parameter p is correct. v is a recognition result. $p \in \text{params}(AD) \cup \{aTask\}$. Ambiguous results not resulting from speech recognition, *e.g.*, Frankfurt am Main *vs.* Frankfurt an der Oder, would yield multiple **requestValue(p=v)** primitives.

requestValueABC(p): system requests the spelling of the value of parameter p .

requestParam(p=v): system asks whether the value v is a value for parameter p .

evaluate(p=v) : system acknowledges value v of parameter p .

promise(p=v): system promises to attempt to answer the user's request. $p \in \text{params}(AD) \cup \{aTask\}$. $v \in \{\text{pleaseWait}\}$. Only used after **navigate()**, **requestParam()** or **requestAlternative()** if the user has to wait long for a reply.

inform(aTask=v): system informs about the acquired database results. $v \in \text{activeTask}(AD) \cup \{\text{tooMany}, \text{zero}\}$. If $v = \text{activeTask}(AD)$, there are several answers, if $v = \text{tooMany}/\text{zero}$, there are either too many answers to be enumerated or zero answers.

inform(aTask=n): system presents the n 'th answer to the query t . $n \geq 0$

informAlternative(p): system informs that there are several possible values for p . $p \in \text{params}(AD) \cup \{aTask\}$. $v \in \{\text{tooMany}, \text{null}\}$. If $v = \text{tooMany}$, there are too many alternatives to be enumerated. $v = \text{null}$, means that v is uninstantiated, *not* that there are zero alternatives.

informAlternative(p=v): system informs that a possible value of p is v . $p \in \text{params}(AD) \cup \{aTask\}$.

informNegative(p): system informs that the user misrecognized something. $p \in \text{params}(AD) \cup \{aTask\}$.

informPositive(p): system informs that the user recognized something correctly. $p \in \text{params}(AD) \cup \{aTask\}$.

withdraw(p): system withdraws from dialogue for reason $p \in \{\text{error}\}$ before it has started negotiations.

withdrawOffer(aTask=v): system withdraws an offer for reason $v \in \{\text{error}\}$.

withdrawPromise(aTask=v): system withdraws a promise for reason $v \in \{\text{error}\}$.

In Section 3, we present several sample instantiations of the primitives.

2.3.2 REC-Primitives

Our current REC-primitives:

requestParam(p): user requests which parameter the system requested. $p \in \text{params}(AD) \cup \{\text{null}\}$.

requestAlternatives(p): user requests possible values for parameter p .

requestConfirm(aTask=n): user asks system to confirm an answer that it has given, *e.g.*, "Was the first answer \$30?" $0 \leq n < \text{no of query results}$.

informValue(p=v): user provides value v for parameter p . p was requested.²

informExtraValue(p=v): user provides value v for parameter p . p was *not requested* in the preceeding system utterance.

informValueABC(p=v): user spells the value v of parameter p . The spelling is expanded by synsem and expansions are presented to the dialogue manager.²

informPositive(p=v): user confirms that the value of parameter p is v . $p \in \text{params}(AD) \cup \{aTask\}$.

informNegative(p=v): user disconfirms that the value of parameter p is v . $p \in \text{params}(AD) \cup \{aTask\}$.

correctValue(p=v): user corrects a misrecognized value. Often used together with **informNegative**. For example, "Hamburg, not Homburg."²

informGarbage(p): user says something but recognizer and/or synsem could not make sense out of it.

changeValue(p=v): user changes the value of parameter p to v instead of v' .²

repeatValue(p=v): user repeats the value v of parameter p .²

correctParam(p=v): user corrects that v is the value of p , not p' .

disambiguate(p=v): user chooses v as the value of p when presented with a choice between several values for p . $p \in \text{params}(AD) \cup \{aTask\}$.

²The pragmatic interpreter instantiates v .

rejectValue(p=v): the user has been given a series of alternatives and chooses $p=v'$. Primitive is combined with **disambiguate(p=v')**.

navigate(aTask=v): user navigates in the query results. $v \in \{\text{forward, backward, repeat, } n\}$ where $0 \leq n < \text{no of query results}$.²

rejectRequest(p=v): user ignores or does not hear the system request. $v \in \{\text{null, didNotHear}\}$.

rejectOffer(aTask=v): user ignores or does not hear the system offer. $v \in \text{tasks}(AD) \cup \{\text{null, didNotHear}\}$.

evaluate(t=v): user evaluates an answer she has received. $v \in \{\text{positive, neutral, negative, cancel}\}$. **cancel** is used to end the current dialogue after at least one answer has been given and start a new one without calling again.

promise(p): user promises to find a value for p .

withdrawAccept(aTask=v): user withdraws from the conversation for reason $v \in \{\text{cancel, hangup}\}$. With **cancel**, the user ends the current dialogue before an answer has been given and starts a new task without calling again.²

withdrawPromise(p=v): user withdraws a promise to provide a value for reason $v \in \{\text{cancel, hangup}\}$.²

withdrawRequest(p=v): user withdraws a request. $p \in \text{params}(AD) \cup \{\text{forward, backward, repeat, and } n\}$.²

null(): returned to the dialogue manager if the user does not say anything and is *not expected* to say anything, *e.g.*, after a greeting or promise.

In Section 3, we present several sample instantiations of the primitives.

2.4 Dialogue Engine

The dialogue engine (Hagen, 1999) consists of a reasoning engine and several knowledge sources: An AD defines an application's data-needs, a dialogue grammar defines how a dialogue may proceed at the level of speech acts, and a dialogue history is a dynamically growing parse tree of an on-going dialogue with respect to the dialogue grammar. Other knowledge sources may be required, for instance, recognition confidence or disambiguation of city names.

The dialogue engine calculates the next turn by consulting and combining information from the knowledge sources. It consults with the dialogue history and the dialogue grammar in order to calculate which speech acts may continue a dialogue. Speech acts have no propositional content, thus in the context of the current dialogue history and

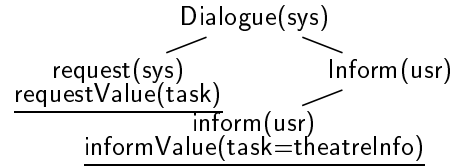
the state of the application description, they are translated into dialogue primitives, which have content, for example, the name of a parameter and a potential value for this parameter. Here we will walk through an example of how some primitives are calculated in a simple question-answer dialogue.

Example: For our example we will use the **AD** in Figure 2. Assume that the task has already been negotiated and set to theatre information (*i.e.*, $\text{activeTask}(AD) = \text{theatreInfo}$), *i.e.*, the system needs to acquire the name of the theatre and the name of the city. All other nodes in the AD are *closed* since they are not relevant to this task.

The **speech act grammar** used in our system is presented in Appendix A but we will use a trivial grammar for the example. It can account for simple question-answer dialogues where a request from the system (sys) is followed by an inform from the user (usr). The system can respond to the inform with a sub-dialogue:³

Dialogue(sys) \rightarrow (request(sys) + Inform(usr))*
Inform(usr) \rightarrow inform(usr) + [Dialogue(sys)]

The **dialogue history** reflects all previous negotiations (here: task theatreInfo).



The next turn can be rooted in either the **Inform(usr)** after the **inform(usr)** or in the **Dialogue(sys)** after **Inform(usr)**.

With all the above knowledge sources in place, the calculation of the next dialogue turn can start:

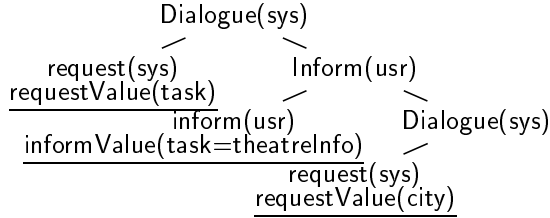
1. The last speech act in the dialogue history gives us a starting point in the grammar, thus moving forward from **inform(usr)**, the next atomic speech act is **request(sys)**—either as a flat structure (*i.e.*, **request(sys)** off **Dialogue(sys)**) or in a sub-dialogue (*i.e.*, **Dialogue(sys)+request(sys)** off **Inform(usr)**).

2. Knowing that the system can request something, the dialogue engine consults with the AD for *what* the system can ask about. The flat structure (**request(sys)**) represents negotiation of the task but since we assume that negotiation of the task is complete (*i.e.*, $\text{Status}(\text{theatreInfo}) = \text{active}$), this speech act is not interpreted into a prim-

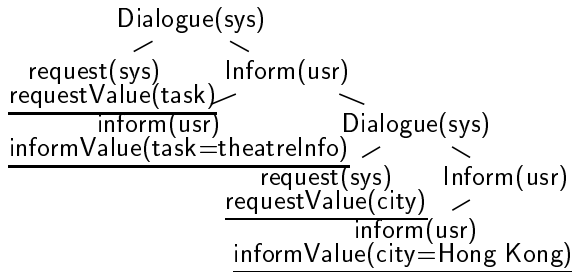
³The star (*) means that a dialogue may contain several **request(sys) + Inform(usr)** sequences. Lower-case speech acts are atomic, while others are complex. The dialogue in square brackets ([]) is optional.

itive. Next we consider the sub-dialogue structure. Both children of `theatreInfo` are *open* (*i.e.*, they have not been negotiated yet) thus the system randomly chooses to pursue `city` whose state is changed to *topic*. The speech act and the parameter are combined into the primitive `requestValue(city)`—request a value for the parameter `city` (*e.g.*, “In which city is the theatre?”). We chose to use the sub-dialogue structure instead of the flat structure to represent negotiation of parameter values since they are subordinate to the task in the sense that the task dictates which parameter values are needed. This is also the case for the real grammar (Appendix A).

3. The primitive `requestValue(city)` is added to the dialogue history:



4. Starting from `request(sys)`, the grammar states that `inform(usr)` (*i.e.*, `Inform(usr)` + `inform(usr)`) is the next speech act in the dialogue. `requestValue(city)` was the last primitive spoken. Reasoning that a user-inform in response to a system `requestValue` should involve the same parameter as the system’s `requestValue`, the information is combined to form the primitive `informValue(city)`, *i.e.*, the user should respond to the system request with a value for the parameter `city`. Let’s assume that the user replied “Hong Kong”, thus the dialogue history is expanded:



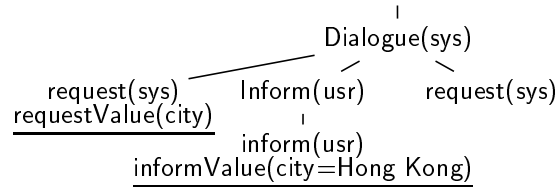
5. Starting from `inform(usr)`, the grammar returns `request(sys)` and `Dialogue(sys)+request(sys)`. Since a recognition result is available from the previous turn, the engine checks its recognition confidence. If it is high, it would consider the negotiation of `city` finished, change its state to *closed*, and discard `Dialogue(sys)+request(sys)` since there is nothing to be requested about a closed parameter. It would translate `request(sys)` into `requestValue(theatre)` since `theatre` is the only remaining

open parameter.

If confidence is low, the dialogue engine may decide to ask the user to confirm the recognized value. In which case, `Dialogue(sys)+request(sys)` would be interpreted into `requestConfirm(city=Hong Kong)`. Whether `request(sys)` would be interpreted or not depends on the dialogue strategies chosen by the service designer (see Sections 3 and 4).

If confidence is extremely low, the dialogue engine may decide to repeat the question. In which case, `request(sys)` would be interpreted into `requestValue(city, 2)`, while the sub-dialogue structure would be discarded.

6. Any interpretation of the flat structure would result in the following addition to the last `Dialogue(sys)` in the dialogue history.



Our example shows how a speech act can result in several primitives depending on the context and thus how the dialogue manager dynamically reacts to external events. Although this brief description may not show it, our dialogue manager can handle mixed initiative dialogue (Hagen, 1999). In (Hagen, 1999), we also present our theory of taking, keeping, and relinquishing the initiative.

Heisterkamp and McGlashan (1996) presented an approach that uses a similar division of functionality as we do: task (=application), contextual (=synsem + pragmatic), and pragmatic interpretation (=dialogue engine). They also use abstract parameterized units similar to ours, but they do not use a speech act grammar to calculate the units. Rather, they map contextual functions onto dialogue goals, *e.g.*, the function `new_for_system(goalcity:munich)` introduces the dialogue goal `confirm(goalcity:munich)`. In terms of our primitives this could be expressed as `requestConfirm()` follows `informValue()`. We choose not to start our modelling at this level since we want to be able to vary what follows `informValue()`, *e.g.*, `requestConfirm()`, `requestValueABC()`, or `evaluate()`.

3 Primitives in Use

Conceptually, GEN-primitives are calculated first and then a bag of possible responses (REC-primitives). One dialogue primitive corresponds to one information unit or communicative goal,

GEN-Primitive	REC-Primitives
requestValue(film)	informValue(film) rejectRequest(film) withdrawAccept(aTask=hangup) withdrawAccept(aTask=cancel)
requestConfirm (theatre=Ridge)	informPositive(theatre=Ridge) informNegative(theatre=Ridge) rejectRequest(theatre=Ridge) withdrawAccept(aTask=hangup) withdrawAccept(aTask=cancel)

Table 1: REC-primitives calculated in response to two GEN-primitives in Dialogue 1.

e.g., in an information retrieval setting: providing or requesting one piece of information. Primitives can be used individually or combined to account for more complex dialogue. Whether and how they are combined depends on the dialogue strategies specified by the service designer. In this and the following section, we will examine several such strategies and show how the primitives are combined to achieve them.

3.1 Question-Answer Dialogue

In the simplest case, the service designer wants a strickt question-answer dialogue:⁴

Dialogue 1: Question-Answer

Sys: “Which film do you want to see?”
requestValue(film)
Usr: “The Matrix. At the Ridge.”
Int: informValue(film=Matrix)
Sys: “Which theatre?”
requestValue(theatre)
Usr: “Ridge. R I D G E.”
Int: informValue(theatre=Ridge)
Sys: “Did you say The Ridge?”
requestConfirm(theatre=Ridge)
Usr: “Yes. R I D G E.”
Int: informPositive(theatre=Ridge)

For this type of dialogue, only the REC-primitives representing direct answers, **rejects**, and **withdraws** are calculated. In Table 1, we present those calculated in response to the first and the third system turn. We see that, after requestValue(film), only informValue(film) is calculated and the pragmatic interpreter has no chance to detect “At the Ridge” (even if synsem parsed it correctly) since there is no informExtraValue(theatre) available to map it onto. Similarly, after requestConfirm(theatre=Ridge) only informPositive(theatre=Ridge) and informNegative(theatre=Ridge) are available and “R I D G E” cannot be detected since there is no informValueABC(city) primitive present.

⁴In the sample dialogues, ‘Sys’ means system turn, ‘Usr’ means user turn, and ‘Int’ means primitives recognized and sent back to the dialogue engine from the pragmatic interpreter.

GEN-Primitive	REC-Primitives
requestValue(film)	informValue(film) rejectRequest(film) informExtraValueValue(time) informExtraValue(theatre) informExtraValue(city) informExtraValue(noOfTickets) informExtraValue(date) withdrawAccept(aTask=v) ^a
requestConfirm (theatre=Ridge)	informPositive(theatre=Ridge) informNegative(theatre=Ridge) rejectRequest(theatre=Ridge) informExtraValue(time) informExtraValue(city) informExtraValue(noOfTickets) informExtraValue(date) withdrawAccept(aTask=v) ^a

^a $\forall v \in \{\text{cancel, hangup}\}$

Table 2: REC-primitives calculated in response to two GEN-primitives in Dialogue 2.

3.2 Over-Answering

In our experience, users frequently provide more information than explicitly asked for, thus a more flexible dialogue strategy would be to allow over-answering and Dialogue 1 could have developed as follows:

Dialogue 2: Over-Answering

Sys: “Which film do you want to see?”
requestValue(film)
Usr: “Matrix. At the Ridge. R I D G E.”
Int: informValue(film=Matrix)
+ informExtraValue(theatre=The Ridge)
Sys: “Did you say The Ridge?”
requestConfirm(theatre=Ridge)
Usr: “Yes, and I want the late show.”
Int: informPositive(theatre=Ridge)
+ informExtraValue(time=9PM)

In Table 2, we present the REC-primitives calculated in response to the same system turns as in Dialogue 1. In Dialogue 2, only over-answering of requestValue() primitives were allowed, thus “R I D G E” could still not be accounted for.

3.3 Complex Mixed Initiative

Here we consider the most complex dialogue strategy that we can currently offer: The system is able to account for complex mixed initiative dialogue (at least from a dialogue point of view), *i.e.*, the user can request clarifications, over-answer, change values, repeat values, correct values, spell values, and reject requests as she pleases.

Dialogue 3: Complex Mixed Initiative

Sys: “Which film do you want to see?”
requestValue(film)
Usr: “Sorry, did you ask for the time?”
Int: requestParam(time)
Sys: “No. Which film do you want to see?”
informNegative(time) + requestValue(film, 2)

Dialogue 3 cont'd.

Ustr: "Matrix. At the Ridge."
 Int: informValue(film=Matrix)
 + informExtraValue(theatre=The Ridge)
 Sys: "Did you say The Ridge?"
 requestConfirm(theatre=Ridge, 1)
 Ustr: "Sorry, I didn't hear that."
 Int: rejectRequest(theatre=didNotHear)
 Sys: "Did you say The Ridge?"
 requestConfirm(theatre=Ridge, 2)
 Ustr: "Yes, The Ridge. R I D G E."
 Int: informPositive(theatre=Ridge)
 + repeatValue(theatre=Ridge)
 + informValueABC(theatre=Ridge)
 Sys: "Ok. What time?"
 evaluate(theatre=Ridge)
 + requestValue(time)
 Ustr: "I don't know. What are the alternatives?"
 Int: requestAlternatives(time)
 Sys: "18:30 or 21:00."
 informAlternative(time=18:30)
 + informAlternative(time=21:00)
 Ustr: "Ok, two tickets for the late show tomorrow."
 Int: evaluate(time=neutral)
 + informExtraValue(noOfTickets=2)
 + informValue(time=21:00)
 + informExtraValue(date=July 4)
 Sys: "Did you say two tickets?"
 requestConfirm(noOfTickets=2)
 Ustr: "Yes, but I change to the early show."
 Int: informPositive(noOfTickets=2)
 + changeValue(time=18:30)

In Table 3, we present the REC-primitives calculated in response to two system utterances.

3.4 Multi-Functional Turns

It has been argued that speech act grammars cannot be used to describe dialogue since utterances can be multi-functional or encode more than one speech act; Speech act grammars can typically be in only one state at a time, thus they cannot capture this phenomenon (Levinson, 1981). In an information retrieval setting such situations occur, for example, when users disregard the system utterance and provide unrelated information or when a recognition mistake occurred and the system asks for confirmation. Instead of answering yes or no, users frequently answer with the correct value, which implicitly disconfirms the previous value:

Dialogue 4: Multi-Functional Utterances

Sys: "How many tickets?"
 requestValue(noOfTickets)
 Ustr: "I want tickets for July 4."
 Int: rejectRequest(noOfTickets)
 + informExtraValue(date=July 3)
 Sys: "Did you say July 3?"
 requestConfirm(date=July 3)
 Ustr: "Tomorrow!"
 Int: informNegative(date=July 3)
 + correctValue(date=July 4)

In the first utterance, the user both ignores the system utterance and provides some information. In the second one, she negated and corrected the system suggestion with a single word.

GEN-Primitive	REC-Primitives
requestValue(film)	informValue(film) informValueABC(film) requestAlternatives(film) promise(film) rejectRequest(film=v) ^a informGarbage(film) requestParam(p) ^b informExtraValue(p) ^b informValueABC(p) ^b repeatValue(p) ^c changeValue(p) ^c withdrawAccept(aTask=v) ^d
requestConfirm (theatre=Ridge)	informPositive(theatre=Ridge) repeatValue(theatre=Ridge) informNegative(theatre=Ridge) correctValue(theatre) informValueABC(theatre) rejectRequest(theatre=v) ^a informGarbage(theatre) informExtraValue(p) ^b informValueABC(p) ^b repeatValue(p) ^c changeValue(p) ^c withdrawAccept(aTask=v) ^d

^a $\forall v \in \{\text{null}, \text{didNotHear}\}$

^b $\forall p \in \text{openParams}(AD)$

^c $\forall p \in \text{closedParams}(AD)$

^d $\forall v \in \{\text{cancel}, \text{hangup}\}$

Table 3: REC-primitives calculated in response to two GEN-primitives in Dialogue 3.

Since we are not using the speech act grammar directly and instead interpret the speech acts into a bag of primitives, we can assign as many primitives to an utterance as necessary and are not bound by the states dictated by a grammar. This aspect of our approach becomes even more interesting when the system combines several primitives in its utterance (Section 4).

4 Dialogue Strategies

Although, the procedure outlined in Section 2.4, only shows how to calculate one primitive per system turn, the approach is, of course, not limited to this. The service designer can decide to employ mixed initiative dialogue strategies for the system utterances as well, for example, requesting or confirming several values at once or implicitly confirming values. The dialogue strategies for system utterances include choosing nodes in the application description, dealing with speech recognition results, or dealing with ambiguous data from other knowledge sources. Here we present a few examples of how the dialogue manager would combine hypotheses (for more information see (Hagen, 2001)).

4.1 Confirmation Strategies

We illustrate implicit and multiple confirmation, *i.e.*, the system realizes `requestValue` and `requestConfirm` or multiple `requestConfirm` primitives in one utterance:

Dialogue 5: Confirmation Strategies

Sys: "Which showing of The Matrix do you want?"
`requestValue(time)`
`+ requestConfirm(film=Matrix)`
 Usr: "(No.) Buena Vista!"
 Int: `informNegative(film=The Matrix)`
`+ correctValue(film=Buena Vista)`
`+ rejectRequest(time)`
 Sys: "Which showing of Buena Vista do you want?"
`requestConfirm(film=Buena Vista)`
`+ requestValue(time)`
 Usr: "The late show. Tomorrow."
 Int: `informPositive(film=Buena Vista)`
`+ informValue(time=21:00)`
`+ informExtraValue(date=8 October)`
 Sys: "Did you say 21:00 today?"
`requestConfirm(time=21:00)`
`requestConfirm(date=October 7)`
 Usr: "No. Tomorrow."
 Int: `informPositive(time=21:00)`
`+ informNegative(date=October 7)`
`+ correctValue(date=October 8)`

For the first two utterances, the system has a recognition result for the parameter `film` with a low recognition score. Consequently, it calculates `requestConfirm(film=Matrix/Buena Vista)`. Additionally, there are still *open* parameter nodes in the AD, thus the dialogue engine picks one (either at random or if the service designer has ordered them, the next one) and calculates a `requestValue` primitive, here `requestValue(time)`. If the service designer allows implicit confirmation, the two primitives are combined and uttered together in one turn. If the service designer does not allow implicit confirmation, the dialogue engine continues the dialogue with the topic that has already been introduced, *i.e.*, `requestConfirm(film=Matrix/Buena Vista)`.⁵

For its last utterance, the system has two recognition results with a low recognition score, thus for each one of them it calculates a `requestConfirm` primitive. If the service designer allows multiple confirmations, they are combined and realized as one utterance. If not, the dialogue engine chooses `requestConfirm(time=21:00)`, since this topic was introduced first. If topics are introduced in the same utterance, it picks one at random.

4.2 AD Based Strategies

When requesting parameter values from the user, the system consults the application description for

⁵This is a conceptual account. In the implementation, the `requestValue` primitive would not be calculated at all, if the service designer does not allow implicit confirmation.

open nodes. If there are several *open* nodes, the dialogue manager can decide to keep the initiative and produce several primitives, which can be combined into one turn. If the nodes are joined with an or-relation, the text generator would translate the primitives into an utterance offering alternative ways of entering the same information. For example, "Please tell me the show time or early or late show." (`requestValue(time)` + `requestValue(namedTime)`). If the nodes are joined with an and- or a has-a relation, the text generator would translate the primitives into an utterance requesting several different pieces of information. For example, "What is the name of the city and the theatre?" (`requestValue(city)` + `requestValue(theatre)`).

As seen in the application descriptions there may be several ways of acquiring a particular value *e.g.*, `date` and `time` in Figure 2. If a parameter value is recognized with a low score, the service designer can decide whether the system shall continue processing the original parameter or whether it shall switch to one of the alternative ones. Thus after a bad recognition of `date`, the system can switch strategy and request `weekDay` and `week` instead.

Which strategies to follow is decided by the service designer through a set of switches in the dialogue strategies specification file (Figure 1).

5 Pragmatic Interpreter

After synsem interpretation, the user utterance must be mapped onto dialogue primitives. A bag of REC-primitives is calculated for each user utterance and the pragmatic interpreter must assure that the utterance is mapped onto primitives in this bag. There is always a mapping. The `reject` and `withdraw` primitives are always part of the bag thus in the worst case, the user utterance would be mapped onto one of these.

Since primitives in their uninstantiated form are application independent, we can develop generic rules for this mapping. In other words, the rules define how the dialogue strategies presented in Section 3 are mapped onto primitives and how we account for several primitives per utterance.

A rule has the form: $\text{GEN-Primitives} \wedge \text{user utterance} \Rightarrow \text{REC-primitives}$. In Table 4, we present two rules for implicit confirmation. The first one corresponds to the first sys/usr pair in Dialogue 5. The user responds with a new value v_3 (Buena Vista) for p_2 (film) in `requestConfirm($p_2=v_2$)` and thereby disconfirms v_2 (Matrix) and rejects the request for a value for p_1 (time) in `requestValue(p_1)`. The second rule corresponds to

GEN-Primitives	Input	REC-Primitives
requestValue(p_i) $\forall 1 \leq i \leq \max_i$ requestConf.($p_j=v_j$) $\forall 1 \leq j \leq \max_j$	(no) $p_j=v_i$, $v_i \neq v_j$, $\forall j$ $1 \leq j \leq$ $k \leq \max_j$	informNeg.($p_j=v_j$) $\forall j 1 \leq j \leq k$ correctVal.($p_j=v_i$) $\forall j 1 \leq j \leq k$ informPos.($p_j=v_j$) $\forall j k \leq j \leq \max_j$ rejectRequest(p_i) $\forall 1 \leq i \leq \max_i$
requestValue(p_i) $\forall 1 \leq i \leq \max_i$ requestConf.($p_j=v_j$) $\forall 1 \leq j \leq \max_j$	$p_i=v_i$, $\forall i$ $1 \leq i \leq$ $k \leq \max_i$	informVal.($p_i=v_i$) $\forall i 1 \leq i \leq k$ informPos.($p_j=v_j$) $\forall 1 \leq j \leq \max_j$ rejectRequest(p_i) $\forall i k \leq i \leq \max_i$

Table 4: Mapping of user input onto REC-primitives. $p=v$ means that the user provided value v for param p .

the second sys/usr pair in Dialogue 5. The user provides value v_1 (the late show) for p_1 (time) in requestValue(p_1) and thus confirms v_2 (Buena Vista) in requestConfirm($p_2=v_2$). For instantiation of the primitives, see Dialogue 5. Here, we only presented two examples. Similar rules were developed for all our primitives and dialogue strategies (see (Hagen, 2001)).

One reviewer asked whether we can modify the approach such that expectations can be overridden if there is sufficiently good information from the synsem module. The short answer is that we could (re-)calculate the primitives pretending that the service designer allowed mixed initiative regardless of the dialogue strategies actually chosen. We, however, think it is important to give her the right to decide. For example, if she has decided that over-answering is allowed, informExtraValue() primitives for all parameters whose status is still *open* would be calculated and thus there is nothing to override. If, however, the service designer has decided that over-answering is not allowed, we assume that she had good reasons for doing that and the dialogue manager will not try to overrule this decision.

6 Conclusion

We have presented some results from our research on spoken dialogue management. We concentrated on how to dynamically calculate a collection of predictions for how to continue a dialogue (dialogue primitives), how to account for different dialogue strategies and utterances with several communicative goals through combinations of primitives, and how to map the user utterances onto primitives. The approach has been implemented and tested in several prototype systems,

e.g., horoscope, movie, and telephone rate service (Feldes et al., 1998).

Dialogue grammars have previously been used to manage dialogue (Bunt, 1989; Bilange, 1991; Traum and Hinkelman, 1992; Jönsson, 1993; Mast et al., 1994; Novick and Sutton, 1994; Chino and Tsuboi, 1996), but we are not aware of an approach where speech acts are translated into a collection of primitives with propositional content. Previous grammar approaches use the speech acts directly or assume a one-to-one correspondence between utterance and speech act.

Through the natural division of the knowledge into type and content, we have achieved a flexible dialogue manager that adapts to users' behaviour. We can take advantage of the predictive capabilities of speech act grammars and still be able to account for multi-functional utterances.

We have also demonstrated that our approach is flexible: 1. the dialogue engine, the pragmatic interpreter, the primitives and the algorithm for mapping user utterances onto predictions are application and language independent, which makes it easy to reuse our dialogue manager in new applications. and 2. the dialogue manager can easily account for several types of dialogue, *e.g.*, strict question-answer or mixed initiative. We give the service designer the freedom to decide which kind of dialogue she wants—on a high level—and the dialogue manager combines the basic primitives accordingly.

Future work includes empirical testing to verify whether we are calculating appropriate predictions. Also, several aspects of our dialogue grammar have not yet been translated into primitives, for example, the frequent use of assert in natural dialogue. As a wider dialogue coverage is required, we will add primitives accordingly. We are also working on using the primitives as input to a multi-lingual automatic text generation system.

Acknowledgements

The author thanks the three anonymous reviewers for their helpful comments on the first draft of this paper. Financial support from the Norwegian Research Council, project number 116578/410 is greatly appreciated.

References

- J.A. Bateman, B. Magnini, and F. Rinaldi. 1994. The generalized {Italian, German, English} upper model. In *Proc. of the ECAI94 Workshop: Comparison of Implemented Ontologies*, Amsterdam, The Netherlands.

- J.A. Bateman, B. Magnini, and G. Fabris. 1995. The generalized upper model knowledge base: Organization and use. In *Proc. of the Conf. on Knowledge Representation and Sharing*, Twente, The Netherlands.
- E. Bilange. 1991. A task independent oral dialogue model. In *Proc. of the Euro. Conf. of the ACL*, pages 83–87.
- H.C. Bunt. 1989. Information dialogues as communicative action in relation to partner modeling and information processing. In M.M. Taylor, F. Neel, and D.G. Bouwhuis, editors, *The Structure of Multimodal Dialogue*, pages 47–73. North-Holland, Amsterdam.
- J. Caminero-Gil, J. Alvarez-Cercadillo, C. Crespo-Casas, and D. Tapias-Merino. 1996. Data-driven discourse modeling for semantic interpretation. In *Proc. of 1996 Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'96)*, pages 401–404.
- T. Chino and H. Tsuboi. 1996. A new discourse model for spontaneous spoken dialogue. In 1021-1024, editor, *Proc. of the 1996 Intl. Conf. on Spoken Language Processing (ICSLP'96)*.
- S. Feldes, G. Fries, E. Hagen, and A. Wirth. 1998. A novel service creation environment for speech enabled database access. In *Proc. 4th IEEE Workshop on Interactive Voice Technology for Telecommunications Applications (IVTTA'98)*, 29-30 Sept. 1998, Torino, Italy.
- E. Hagen. 1999. An approach to mixed initiative spoken information retrieval dialogue. *User Modeling and User-Adapted Interaction*, 9(1/2):167–213.
- E. Hagen. 2001. *Mixed Initiative Spoken Dialogue Management in Information Systems*. Ph.D. thesis, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada. Jan. 2001 expected.
- P. Heisterkamp and S. McGlashan. 1996. Units of dialogue management. In *Proc. of the 1996 Intl. Conf. on Spoken Language Processing (ICSLP'96)*.
- A. Jönsson. 1993. A dialogue manager using initiative-response units and distributed control. In *Proc. of 6th Euro. Conf. of the ACL*, pages 233–238.
- S.C. Levinson. 1981. Some pre-observations on the modelling of dialogue. *Discourse Processes*, 4:93–116.
- M. Mast, F. Kummert, U. Ehrlich, G.A. Fink, T. Kuhn, H. Niemann, and G. Sagerer. 1994. Prosody takes over: Towards a prosodically guided dialog system. *Speech Communication*, 15(1–2):155–167.
- D.G. Novick and S. Sutton. 1994. An empirical model of acknowledgement for spoken-language systems. In *Proc. of the 32nd Annual Meeting of the ACL*, pages 96–101.
- S. Sitter and A. Stein. 1992. Modelling the illocutionary aspects of information-seeking dialogues. *Information Processing and Management*, 8(2):165–180.
- D. Traum and E. Hinkelman. 1992. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599.
- S. Young, A. Hauptmann, W. Ward, E. Smith, and P. Werner. 1990. High-level knowledge sources in usable speech recognition systems. In A. Waibel and K. Lee, editors, *Readings in Speech Recognition*, pages 538–549. Morgan Kaufman, San Mateo, CA.

A. The Complete Dialogue Grammar

The complete grammar is a slightly modified version of the grammar presented in (Sitter and Stein, 1992). **Notation:** Complex dialogue moves begin with an upper case (*e.g.*, Request); atomic dialogue acts are all lower case (*e.g.*, request). S and K mean seeker and knower. Square brackets ([]) mean optional. X^+ means one or more instances of X. All moves except Inform and Assert are represented by the abstraction Move. Subscript $_i$ means that move and act must be of the same type, *e.g.*, Request and request.

Dialogue(S) \rightarrow (Cycle(S))⁺
 Cycle(S) \rightarrow Request(S), Promise(K), Inform(K), Evaluate(S).
 Cycle(S) \rightarrow Request(S), [Promise(K)], WithdrawRequest(S).
 Cycle(S) \rightarrow Request(S), Promise(K), WithdrawPromise(K).
 Cycle(S) \rightarrow Request(S), RejectRequest(K).
 Cycle(S) \rightarrow Offer(K), Accept(S), Inform(K), Evaluate(S).
 Cycle(S) \rightarrow Offer(K), [Accept(S)], WithdrawOffer(K).
 Cycle(S) \rightarrow Offer(K), Accept(S), WithdrawAccept(S).
 Cycle(S) \rightarrow Offer(K), RejectOffer(S).
 Cycle(S) \rightarrow Withdraw(usr).
 Cycle(S) \rightarrow Withdraw(system).

Inform(K) \rightarrow inform(K), [Dialogue(S)].
 Assert(S/K) \rightarrow assert(S/K), [Dialogue(K/S)].

Move_i(S/K).
 Move_i(S/K) \rightarrow act_i(S/K), [Dialogue(K/S)].
 Move_i(S/K) \rightarrow act_i(S/K), [Assert(S/K)].
 Move_i(S/K) \rightarrow Dialogue(K/S).
 Move_i(S/K) \rightarrow Assert(S/K), [act_i(S/K)].
 Move_i(S/K) \rightarrow Assert(S/K), [Dialogue(K/S)].