D4.3: Bayes Net Prototype - the Hidden Information State Dialogue Manager

Steve Young, Jason Williams, Jost Schatzmann
Matt Stuttle, Karl Weilhammer

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 4.3

4th January, 2006

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	M24 = December 2005
Actual date of delivery	4th January, 2006
Deliverable number	4.3
Deliverable title	D4.3: Bayes Net Prototype - the Hidden Information State Dialogue Manager
Type	Report
Status & version	Final 1.0
Number of pages	33 (excluding front matter)
Contributing WP	4
WP/Task responsible	UCAM
Other contributors	
Author(s)	
EC Project Officer	
Keywords	statistical dialogue modelling; partially observable Markov decision processes (POMDPs)

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, `http://www.talk-project.org`

Contents

Executive summary

Partially observable Markov decision processes (POMDPs) provide a principled mathematical framework for modelling the uncertainty inherent in spoken dialogue systems. However, conventional POMDPs scale poorly with the size of state and observation space. This report describes a variation of the classic POMDP called the Hidden Information State (HIS) model which is a practical implementation of the core Bayes Network underpinning the formulation of dialog as a partially observable Markov decision process. The key idea of the HIS system is that a belief distribution over an extremely large state space can be represented efficiently by grouping states together into partitions. Initially, all states are deemed to be in a single partition with belief unity. As the dialog progresses, the partitions are split and belief is redistributed amongst the splits. Eventually, some partitions become very low in cardinality, and in the limit singletons. Action selection is then dominated by these low cardinality partitions. The overall result is that full-scale POMDP belief maintenance is achieved without ever explicitly calculating the beliefs of (the majority of) irrelevant states. Furthermore, partitions are represented efficiently using tree structures and these tree structures also provide a very natural representation for real-world knowledge.

This report is in two parts. The first part presents the theoretical basis of the HIS model and explains the major algorithms and data structures. This report is also published as a CUED technical report number CUED/F-INFENG/TR.544 (2005). The second part briefly describes the software implementation of the HIS model in the form of a C++ class library.

Part I
The Hidden Information State Approach to Dialogue Management

1 Introduction

The structure of a conventional dialogue system is shown in Fig. 1 both in terms of a block diagram showing the data flow, and an influence diagram showing the dependencies from one time slot (i.e. turn) to the next. The processing involved in a single dialog turn proceeds as follows. A dialog manager generates a prompt to the user in the form of a machine dialog act A_m. This is converted to an acoustic signal Y_m and subsequently interpreted by the user as $\tilde{A}_m$. The user has a state which encodes both a goal to achieve S_u and the dialog history S_d. On receiving, $\tilde{A}_m$ the user updates this state and generates a user dialog act A_u. This is converted to an acoustic signal Y_u and interpreted by the system's speech understanding component to give $\tilde{A}_u$. The dialog system maintains its own view of the world in state variable S_m. The estimate $\tilde{A}_u$ is used to update this estimate of the machine's state and based on this updated estimate, the dialog manager generates a new machine dialog act A_m.

Although greatly simplified, this description of the dialog turn cycle applies to nearly all existing systems. In particular, the information state update approach (ISU) to dialog system design can be viewed as a direct implementation of this model [1].

However, although it is simple and intuitive, this traditional deterministic dialog model has a number of severe weaknesses. Firstly, and crucially, in real systems, the estimate of the user's dialog act $\tilde{A}_u$ will be extremely noisy. Hence, the system state S_m must be updated based on a "best guess" of A_u and since this best guess will often be wrong, the system state can be easily corrupted by erroneous information. This will typically lead to misunderstanding and confusion, requiring a perhaps lengthy recovery dialog to repair it. The incidence of this problem can be reduced by making use of a confidence measure output by the speech understanding component. This measure provides an estimate of $P(\tilde{A}_u|Y_u)$ which is typically compared with a threshold and based on the result either $\tilde{A}_u$ is accepted as true, or it is queried with the user. Unfortunately, however, confidence measures themselves are unreliable, and there is no clear basis for setting the threshold.

A second problem with the traditional architecture is that speech understanding errors are not the only source of uncertainty: the user's goals and intentions are uncertain and can change over time. Thus, a model of the user's goals and intentions must be integrated into the overall dialog management process.

A final problem with the traditional architecture is the determinism itself. In order to implement optimal dialog strategies, a system must predict the future in order to plan for differing eventualities. Since exact prediction is not possible, such plans can only be probabilistic and, as with the use of confidence thresholds, these can only be used in a very crude way by a deterministic decision process. Also, of course, it is very hard to adapt deterministic systems from training data, and in practice, adaptation is limited to manual system tuning following an off-line analysis of system logs. This process is labour intensive and cannot be extended to automatic on-line adaptation.

As has been argued previously, taking a statistical approach to spoken dialog system design provides the opportunity for solving many of the above problems in a flexible and principled way[2]. Early attempts at

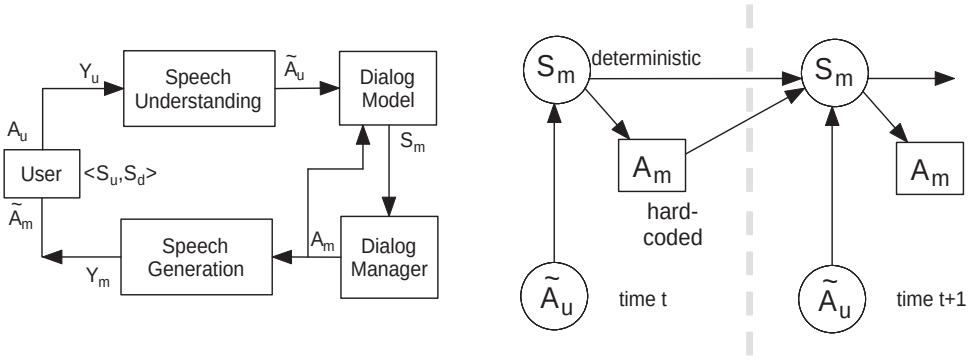

Figure 1: A traditional Spoken Dialog System along with its corresponding influence diagram

using a statistical approach modelled the dialog system as a Markov decision process (MDP)[3]. MDPs provide a good statistical framework since they allow forward planning and hence dialog policy optimisation through reinforcement learning. However, they suffer from a number of problems. Firstly, and crucially, MDPs assume that the machine state is observable. Hence, they cannot account for either the uncertainty in the user state (S_u and S_d in Fig. 1), or the uncertainty in the decoded user's dialog act ($\tilde{A}_u$ in Fig. 1). Secondly, and perhaps less obviously, the MDP approach provides a poor interface for integrating heuristics. The main problem is that heuristics typically involve making hard decisions based on the assumed system state. However, in the case of an MDP, the assumed system state might be incorrect. To deal with this, the state must be expanded to include confidence measures so that the heuristics can deal explicitly with the uncertainty. However, this rapidly leads to an excessively large state space and complex heuristics.

A more general alternative to the fully observable MDP is the Partially Observable MDP (POMDP)[4]. A dialog system based on a POMDP maintains a distribution over all possible states. This distribution is called the *belief state* and dialog policies are based on this belief state rather than the true underlying state. The key advantage of the POMDP formalism is that it provides a complete and principled framework for modelling the main sources of uncertainty. Furthermore, when cast as a so-called Spoken Dialog System POMDP (SDS-POMDP)[5, 6], the framework also allows heuristics to be incorporated in a very simple way since the principal components on which the heuristics depend (e.g. S_u and A_u) are by definition assumed to be true. In computational terms, this means that in an MDP heuristics are executed once per turn but have to be programmed to explicitly take account of uncertainty. In a POMDP, heuristics are much simpler to program because the state is assumed correct. They do however have to be executed many times per turn, once for each possible state value.

The use of POMDPs for any practical system is, however, far from straightforward. Since a belief distribution b of a discrete state S of cardinality $n+1$ lies in real-valued n-dimensional simplex, a POMDP can be thought of as an MDP with a continuous state space $b \in \Re^n$. Thus, assuming that the POMDP machine has a finite set of actions to select from, a POMDP policy is a mapping from partitions in n-dimensional belief space to actions. Not surprisingly these are extremely difficult to construct and whilst exact solution algorithms such as the Witness algorithm [7] do exist, they rarely scale to problems with more than a few states/actions. Fortunately, there are a number of ways of finding approximate solutions which are

sufficiently accurate to yield useful results. Firstly, the planning horizon can be reduced to zero such that actions are selected based on immediate utilities. This reduces the POMDP to a greedy decision-theoretic controller. It does not solve the problem of accounting for the future effects of decisions, but it does allow for all of the various sources of uncertainty to be modelled and accounted for in the decision making process[8]. Secondly, approximate solutions such as grid-based methods[9] and point-based value iteration can be used [10]. These allow problems with several hundreds of state/actions to be handled, and although these are not sufficient by themselves to deal with real world systems, used in conjunction with state-mapping schemes such as the summary-POMDP method, they offer promise of a way forward[11].

Whatever approach is taken to the construction of policies, there is an other fundamental barrier to using POMDPs in spoken dialog systems. Real systems deal with real-world knowledge which is complex, hierachical, and multi-valued. The potential state-space of even a simple travel booking system is enormous. Furthermore, dialog acts cannot easily be enumerated as a simple finite set. The types of act (*request*, *inform*, etc) are easily enumerated, but the arguments to such acts (names of places, prices, dates, etc) are not so simple. Whereas research into MDPs was able to side-step this problem on the grounds that only a few global indicators needed to be modelled[12, 13], a central claim of the POMDP approach is that it is truly holistic and, in particular, propositional content should not be ignored. Thus, whilst POMDPs provide a theoretical framework for modelling complete dialog systems, what is also needed in practice is a framework which can integrate the applicable knowledge representations with the appropriate statistical models.

This report describes the development Bayes Network framework which can support POMDP's with very large hierarchical state spaces. It is inspired by the information state approach to dialog system implementation, and hence it is called the Hidden Information State (HIS) framework for statistical dialog systems. The key idea of the HIS system is that a belief distribution over an extremely large state space can be represented efficiently by grouping states together into partitions[14, 15, 16, 17]. Initially, all states are deemed to be in a single partition with belief unity. As the dialog progresses, the partitions are split and belief is redistributed amongst the splits. Eventually, some partitions become very low in cardinality, and in the limit singletons. Action selection is then dominated by these low cardinality partitions. The overall result is that full-scale POMDP belief maintenance is achieved without ever explicitly calculating the beliefs of (the majority of) irrelevant states. Furthermore, partitions are represented efficiently using tree structures and these tree structures also provide a very natural representation for real-world knowledge.

The remainder of this report is structured as follows. Section 2 briefly reviews the general framework of the Spoken Dialog System POMDP (SDS-POMDP) and section 3 explains how the HIS system fits into this framework. Section 4 then describes the implementation of the HIS system in some detail. Section 5 presents the current greedy theoretic policy implementation, and finally, section 6 discusses future work and conclusions.

2 The SDS-POMDP

The aim of this section is to review the basic POMDP equations and then present a factored form called the SDS-POMDP which is suitable for spoken dialogue systems[5, 6]. This lays the foundation for describing the HIS model in the following section.

2.1 Review of POMDPs

Formally, a POMDP is defined as a tuple $\{S, A_m, T, R, O, Z, \lambda, b_0\}$ where S is a set of states; A_m is a set of actions that the machine may take; T defines a transition probability $P(s'|s, a_m)$; R defines the expected (immediate, real-valued) reward $r(s, a_m)$; O is a set of observations; Z defines an observation probability $P(o'|s', a_m)$; λ is a geometric discount factor $0 \leq \lambda \leq 1$; and b_0 is an initial belief state $b_0(s)$.

The POMDP operates as follows. At each time-step, the machine is in some unobserved state $s \in S$. Since s is not known exactly, a distribution over states is maintained called a "belief state," b, with initial belief state b_0. Thus, the probability of being in state s given belief state b is $b(s)$. Based on the current belief state b, the machine selects an action $a_m \in A_m$, receives a reward $r(s, a_m)$, and transitions to a new (unobserved) state s', where s' depends only on s and a_m. The machine then receives an observation $o' \in O$ which is dependent on s' and a_m. The belief distribution is then updated based on o' and a_m.

The belief update equations are easily derived using Bayes rule:

$$\begin{aligned} b'(s') &= P(s'|o', a_m, b) \\ &= \frac{P(o'|s', a_m, b)P(s'|a_m, b)}{P(o'|a_m, b)} \\ &= \frac{P(o'|s', a_m, b)\sum_{s\in S} P(s'|a_m, b, s)P(s|a_m, b)}{P(o'|a_m, b)} \\ &= \frac{P(o'|s', a_m)\sum_{s\in S} P(s'|a_m, s)b(s)}{P(o'|a_m, b)} \\ &= k \cdot P(o'|s', a_m)\sum_{s\in S} P(s'|a_m, s)b(s) \end{aligned} \quad (1)$$

where k is a normalising constant. In equation 1, the summation uses the transition probability to predict each next state s' as an expectation wrt to the belief state over preceding states. The observation term before the summation weights the prediction for each new state s' based on the likelihood that the most recent observation o' could have been generated from s'.

Note that the action taken by the machine at each time step depends on the complete distribution b. This is often initially a flat distribution reflecting ignorance. At each time-step, the belief state distribution b is updated based on the new observation, and typically this will result in the distribution "sharpening" around specific states.

At each time step t, the machine receives a reward $R(b_t, a_{m,t})$ based on the current belief state b_t and the selected action $a_{m,t}$. The cumulative, infinite horizon, discounted reward is called the *return* and it is given by:

$$R = \sum_{t=0}^{\infty} \lambda^t R(b_t, a_{m,t}) \quad (2)$$

$$= \sum_{t=0}^{\infty} \lambda^t \sum_{s\in S} b_t(s) r(s, a_{m,t}). \quad (3)$$

Each action $a_{m,t}$ is determined by a policy $\pi(b_t)$ and it is the goal of the machine to find the policy π^* which maximises the return. Such a policy is called an optimal policy. Since belief space is a real-valued simplex, the policy can be viewed as a partitioning of belief space into regions, where each region corresponds to the single unique action which should be taken if the current belief state lies in that region.

Finding the optimal policy involves using the transition matrix to predict the reward expected from each state for each possible machine action. This is very similar to the forward-backward algorithm of E-M and for regular fully observed Markov Decision Processes, it is essentially a dynamic programming search over a discrete state space. POMDPs solutions are much more complex, however, because the state space is effectively continuous. As mentioned in the introduction, exact solution algorithms do exist (e.g. see the Witness Algorithm [7, 4]) but they can only handle very small problems. Fortunately, approximate solutions can handle significantly larger problems (e.g. Perseus [10]).

For cases where the model is unknown or there is insufficient data to estimate accurately, on-line learning techniques analogous to Q-learning are also possible. For example, active learning can be used to simultaneously update an approximate model whilst optimising the return[18].

2.2 The SDS-POMDP: a factored POMDP for spoken dialog systems

Referring back to Fig. 1, it can be seen that the state space represented by the dialog model S_m must entail the user goal and dialog state and since these cannot be observed, S_m must correspond to a distribution over those states. In addition, since the last user act is also uncertain, it is convenient to include it also within the unobserved state space. This suggests that the state space of a POMDP for dialog systems should be factored as follows. First, the unobserved state is factored into 3 components:

$$s = (s_u, a_u, s_d). \tag{4}$$

The system state S_m then becomes the belief state b over s_u, a_u and s_d, i.e.

$$s_m = b(s_u, a_u, s_d). \tag{5}$$

The observation o is the estimate of the user dialog act $\tilde{a}_u$. In the general case this will be an N-best list of hypothesised user acts, each with an associated probability, i.e.

$$o = [(\tilde{a}_u^1, p_1), (\tilde{a}_u^2, p_2), \ldots, (\tilde{a}_u^N, p_N)] \tag{6}$$

such that

$$P(\tilde{a}_u^n|o) = p_n, \qquad n = 1 \ldots N \tag{7}$$

The transition function for an SDS-POMDP follows directly by substituting equation 4 into the regular POMDP transition function and making some reasonable independence assumptions, i.e.

$$\begin{aligned} P(s'|s, a_m) &= P(s'_u, a'_u, s'_d|s_u, a_u, s_d, a_m) \\ &= P(s'_u|s_u, a_m)P(a'_u|s'_u, a_m)P(s'_d|s'_u, a'_u, s_d, a_m) \end{aligned} \tag{8}$$

Making similar reasonable independence assumptions regarding the observation function gives,

$$\begin{aligned} P(o'|s', a_m) &= P(o'|s'_u, a'_u, s'_d, a_m) \\ &= P(o'|a'_u) \end{aligned} \tag{9}$$

This is the *observation model*.

The above factoring simplifies the belief update equation since substituting equation 8 and equation 9 into equation 1 gives

$$
\begin{aligned}
b'(s'_u, a'_u, s'_d) = & \qquad (10)\\
& k \cdot P(o'|a'_u) \sum_{s_u, a_u, s_d} P(s'_u|s_u, a_m) P(a'_u|s'_u, a_m) P(s'_d|s'_u, a'_u, s_d, a_m) b(s_u, a_u, s_d) \\
= & \; k \cdot P(o'|a'_u) P(a'_u|s'_u, a_m) \sum_{s_u} P(s'_u|s_u, a_m) \sum_{s_d} P(s'_d|s'_u, a'_u, s_d, a_m) \sum_{a_u} b(s_u, a_u, s_d) \\
= & \; k \cdot \underbrace{P(o'|a'_u)}_{\substack{\text{observation}\\ \text{model}}} \underbrace{P(a'_u|s'_u, a_m)}_{\substack{\text{user action}\\ \text{model}}} \sum_{s_u} \underbrace{P(s'_u|s_u, a_m)}_{\substack{\text{user goal}\\ \text{model}}} \sum_{s_d} \underbrace{P(s'_d|s'_u, a'_u, s_d, a_m)}_{\substack{\text{dialog}\\ \text{model}}} b(s_u, s_d) \qquad (11)
\end{aligned}
$$

As shown by the labelling to equation 11, the probability distribution for a'_u is called the *user action model*. It allows the observation probability that is conditioned on a'_u to be scaled by the probability that the user would speak a'_u given the goal s'_u and the last system prompt a_m. The *user goal model* determines the probability of the user goal switching from s_u to s'_u following the system prompt a_m. Finally, the *dialog model* represents the transition matrix for the dialog state component. This term provides the primary hook for incorporating heuristics into the system. In particular, it allows information relating to the dialog history to be maintained such as grounding and focus.

3 The Hidden Information State dialog model

Having described the general form of the SDS-POMDP in the previous section, this section derives a specific form of SDS-POMDP called the Hidden Information State model.

Although the factoring introduced in the last section is helpful, the size of the state spaces needed to represent real-world dialog systems would quickly render a direct SDS-POMDP implementation intractable. The dialog state component is computed heuristically and as will be explained later, this results in a relatively small set of dialog states being tracked from turn to turn. However, the user goal and action state components require reasonably accurate distributions to be maintained and this is not easy since the size of the user goal space is enormous and the user actions cannot even be enumerated. The HIS model deals with these two components in different ways.

Consider first the user action model. As shown by equation 11, the user action component of the state space is memoryless, i.e. the value of the previous user action a_u is not required to apply the belief update equation. This means that the distribution for a'_u can be approximated by considering just those user action values which are deemed to have non-zero probabilities in the current turn. These will be precisely those actions which appear in the N-best list of hypotheses from the speech understanding component. To guard against the case of very poor recognition resulting in the correct value of a'_u being dropped from the observation altogether, a *null* action is always included with a floor probability representing all of the user acts not in the N-best list.[1]

To deal with the user goal component, it is necessary to be a little more specific about what is meant by a *user goal*. The initial target of the HIS model is database inquiry applications such as traffic information, tourist information, flight booking, etc. In this context, a user goal is deemed to be a specific entity that

[1]Note that in the context of a POMDP-based spoken dialog system, the terms *user act* and *user action* are synonymous.

the user has in mind. For example, in a tourist information system, the user might be wishing to find a moderately priced restaurant near to the theatre. The user would interact with the system, effectively refining his or her query until an appropriate establishment was found. If the user wished to find an alternative restaurant, or even something different entirely such as the nearest tube station to the restaurant, this would constitute a new goal. In the HIS system, the duration of a dialog is defined as being the interaction needed to satisfy a single goal. Hence by definition, the user goal model simplifies trivially to a delta function, i.e.

$$P(s'_u|s_u) = \delta(s'_u, s_u). \tag{12}$$

Substituting equation 12 into equation 11 gives

$$b'(s'_u, a'_u, s'_d) \quad = \quad k \cdot P(o'|a'_u) P(a'_u|s'_u, a_m) \sum_{s_d} P(s'_d|s'_u, a'_u, s_d, a_m) b(s'_u, s_d) \tag{13}$$

To further simplify belief updating, it will be assumed that at any time t, S_u can be divided into a number of equivalence classes where the members of each class are tied together and are indistinguishable. These equivalence classes will be called *partitions* of user goal space. Initially, all states $s_u \in S_u$ are in a single partition p_0. As the dialog progresses, this root partition is repeatedly split into smaller partitions. This splitting is binary

$$p \rightarrow \{p', p - p'\} \qquad \text{with probability} \qquad P(p'|p). \tag{14}$$

Since multiple splits can occur at each time step, this binary split assumption places no restriction on the possible refinement of partitions from one turn to the next.

Given that user goal space is partitioned in this way, beliefs can be computed based on partitions of S_u rather than on the individual states of S_u. Initially the belief state is just

$$b_0(p_0) = 1. \tag{15}$$

Whenever a partition p is split, its belief mass is reallocated according to equation 14, i.e.

$$b(p') = P(p'|p) b(p) \qquad \text{and} \qquad b(p - p') = (1 - P(p'|p)) b(p) \tag{16}$$

Note that this splitting of the belief mass is simply a reallocation of existing mass, it is not a belief update. It will be referred to as *belief refinement*.

The belief update equation for a partitioned state space is easily derived from the non-partitioned case. Let partition p' consist of states $\{s'_u | s'_u \in p'\}$, then summing both sides of equation 13 over all $\{s'_u\}$ gives,

$$b'(p', a'_u, s'_d) \quad = \quad k \cdot P(o'|a'_u) \sum_{s'_u \in p'} P(a'_u|s'_u, a_m) \sum_{s_d} P(s'_d|s'_u, a'_u, s_d, a_m) b(s'_u, s_d) \tag{17}$$

As a dialog progresses, the user goal partitions are split repeatedly to ensure that everything which has been mentioned so far in the dialog is explicitly represented in the partitions. This being so, it is reasonable to assume that

$$P(a'_u|s'_u, a_m) = P(a'_u|p', a_m) \tag{18}$$

and

$$P(s'_d|s'_u, a'_u, s_d, a_m) = P(s'_d|p', a'_u, s_d, a_m) \tag{19}$$

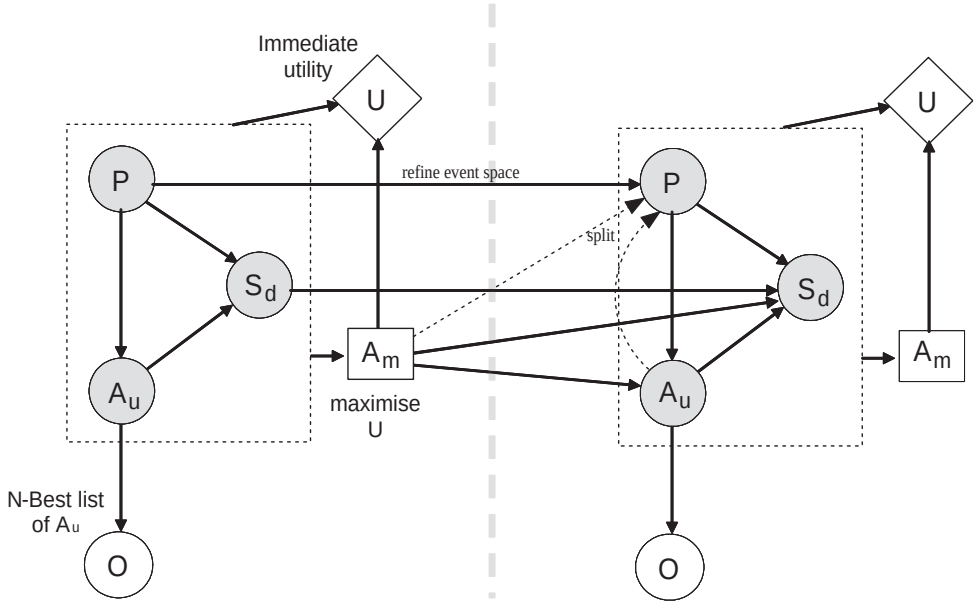

Figure 2: Influence diagram for the Hidden Information State dialog model

Hence, using these simplifying assumptions and equation 16, equation 17 becomes

$$\begin{aligned} b'(p', a'_u, s'_d) &= k \cdot P(o'|a'_u) P(a'_u|p', a_m) \sum_{s_d} P(s'_d|p', a'_u, s_d, a_m) \sum_{s'_u \in p'} b(s'_u, s_d) \\ &= k \cdot P(o'|a'_u) P(a'_u|p', a_m) \sum_{s_d} P(s'_d|p', a'_u, s_d, a_m) b(p', s_d) \\ &= k \cdot \underbrace{P(o'|a'_u)}_{\text{observation model}} \underbrace{P(a'_u|p', a_m)}_{\text{user act model}} \sum_{s_d} \underbrace{P(s'_d|p', a'_u, s_d, a_m)}_{\text{dialog model}} \underbrace{P(p'|p) b(p, s_d)}_{\text{belief refinement}} \end{aligned} \quad (20)$$

where p is the parent of p'. Equation 20 is the belief update equation for the HIS model, it is shown in the form of an influence diagram in Fig. 2. Note that in this diagram the dotted arrows represent the influence of a_m and a_u on the *refinement* of p' but not on its update i.e. they influence the splitting of p' but not its conditional probability.

As shown by the labelling on equation 20, the HIS update equation depends on four probability distributions:

1. *Observation Model* - this is approximated by the N-best probability from the speech understanding component

$$P(o'|a'_u) \approx k' \cdot P(a'_u|o) \quad (21)$$

2. *User Act Model* - this is composed of two parts: the bigram probability of the current user act type given the preceding system act type, and a probability denoting the degree to which the current user

act is consistent with the given partition p'. Thus,

$$P(a'_u|p', a_m) \approx P(\mathcal{T}(a'_u)|\mathcal{T}(a_m))P(\mathcal{M}(a'_u)|p') \quad (22)$$

where $\mathcal{T}(a)$ denotes the *type* of the dialog act a, for example, the type of the act "inform(food=Indian)" is *inform*. There are a total of 12 different dialog act types supported by the HIS model and these are described in detail section 4.3. $\mathcal{M}(a)$ denotes whether or not the dialog act *a matches* the current partition p'. The first component can be estimated from a dialog corpus, the second component is set to 1 if the act matches and zero otherwise.

3. *Dialog Model* - this is entirely heuristic.

$$P(s'_d|p', a'_u, s_d, a_m) = 1 \quad \text{iff } s'_d \text{ is consistent with } p', a'_u, s_d, a_m \quad (23)$$
$$= 0 \quad \text{otherwise} \quad (24)$$

The way that this is computed in the HIS model is described in section 4.5.

4. *Belief Refinement* - this depends on the ontology rules used to define the application domain. User goals are built using probabilistic context free rules, with rule probabilities set *a priori*. If the sequence of rules $r_1, r_2, \ldots, r_k$ is used to split partition p into sub-partition p', the belief refinement probability is

$$P(p'|p) = \prod_{i=1}^{k} P(r_i) \quad (25)$$

where $P(r)$ is the prior probability of rule r. This process is described in more detail in section 4.2.

Having described the mathematical basis of the HIS model, the remainder of this report describes its specific implementation.

4 Implementation of the HIS Model

This section describes a specific implementation of the HIS model. It begins with a high level overview of how the model operates. It then describes each of the main components in more detail.

4.1 Overview of HIS Model Operation

Before describing the details of the HIS system, it will be helpful to give a brief overview of the principle data structures and the overall operation. As shown in Fig. 3, the inputs to the system consist of an observation from the user and the previous system act. The observation from the user typically consists of an N-best list of user acts, each tagged with their relative probability. The user goal is represented by a set of branching tree structures each of which initially consist of just a single node. These tree structures can be grown downwards by applying ontology rules which describe the application domain. For example, there might be a rule which states that a `venue` can be either a `hotel`, a `restaurant` or a `bar`. In each case, the derived venues will have further nodes describing features of that type of venue. Ambiguity is represented by allowing nodes to expand into multiple alternatives. Each distinct tree forms a partition

of user goal space as described in section 3. The initial single tree node represents a single partition with belief unity. As the trees are grown, the partitions are repeatedly split allowing the belief assignment to be refined. Eventually, the hope is that a single complete tree will be formed which represents the actual user's goal and that this tree has a high belief.

The tree growing process is driven entirely by the dialog acts exchanged between the system and the user. Every turn, the previous system act and each input user act is matched against every partition in the branching tree structure. If a match can be found then it is recorded. Otherwise the ontology rules are scanned to see if the tree representing that partition can be extended to enable the act to match. For example, if the act was `request(ensuite)`, and the partition represented the higher level node `venue`, then the venue node would be extended to a hotel node with associated properties, one of which would be `ensuite`. The `request(ensuite)` act would then match. Note however that an ontology rule can be used to extend a specific node just once. This ensures that all partitions are unique and there are no duplicates.

Once the matching and partition splitting is complete, all the partitions are rescanned and where possible each hypothesised input user act is attached to each partition. Similarly the system act is attached to each partition (not shown in the figure). The combination of a partition and an input user act (p, a_u) forms a partial hypothesis and the user act model probability is calculated as in equation 22.

As explained above, partitions are grown based entirely on dialog act inputs. If the user (or the system) mentions a node such as `ensuite` this will cause other nodes to be created. The grounding status of each tree node is recorded in a dialog state data structure. Since the grounding status of a tree node can be uncertain, any (p, a_u) pair can have multiple dialog states attached to it. However, unlike the user act component of the state which is memoryless, the dialog component s_d evolves as the dialog progresses. Thus, at the beginning of each dialog cycle, the various dialog state instances are attached directly to the partitions. Once the input user acts have been attached to the partitions, the current dialog states are extended to represent the new information in the dialog acts. At this point, the dialog state probabilities given by equation 24 are computed. At the end of the turn, identical dialog states attached to the same partition can be merged[2] ready for the next cycle.

Every triple (p, a_u, s_d) represents a single dialog hypothesis h_k. The belief in each h_k is computed using equation 20 and the complete set of values $b(h_k)$ represents the current estimate of the POMDP belief state. However, unlike a full POMDP, the current version of the HIS model does not do forward planning. Instead, each hypothesis h_k is examined and all possible system dialog acts are generated. All of these candidate system dialog acts are collected together into a pool $\{a_m^i\}$ and the conditional utility $U(a_m^i|h_k)$ calculated for each. The expected utility of each candidate can then be computed and the candidate with the maximum utility chosen as the next system move.

$$a_m = \operatorname*{argmax}_i \left\{ \sum_k b(h_k) U(a_m^i|h_k) \right\} \tag{26}$$

The details of how candidate system acts are generated and utilities are calculated are given later in section 5.

4.2 User Goal Trees and Ontology Rules

User goals are represented by a branching tree structure whose hierarchy reflects both the natural structure of the data and a natural order in which to introduce the individual concepts into a conversation. User goal

[2]This merging operation is not essential and is not actually done in the current implementation

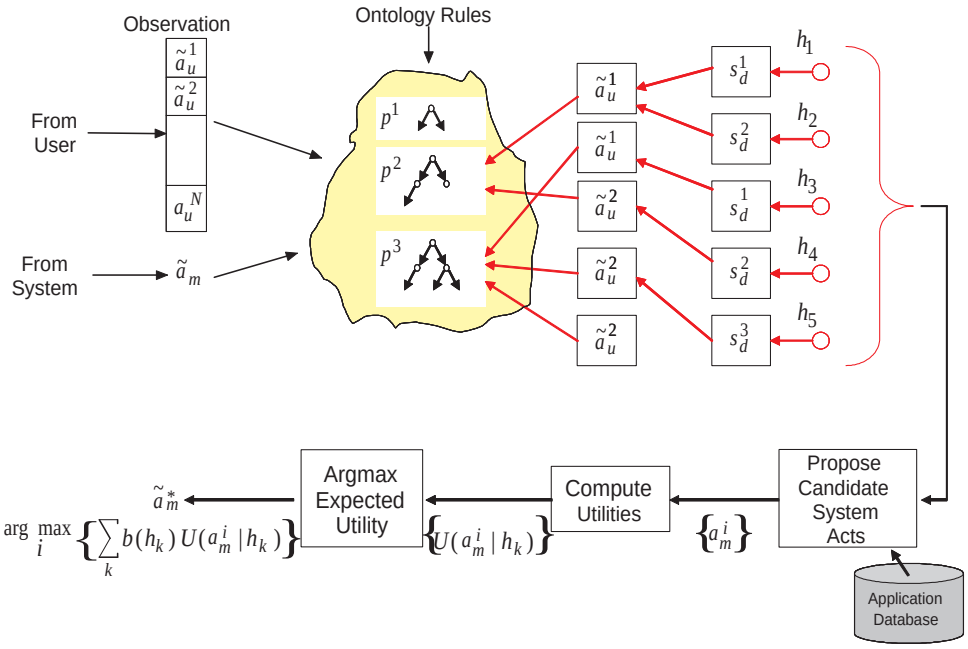

Figure 3: Overview of the HIS System operation

trees are constructed from four types of tree node:

1. class nodes - these have non-terminal offspring. Conceptually a class node represents an instance of a type, and the offspring of the node denote the members of that type.

2. lexical nodes - these have only terminal offspring i.e. atoms.

3. subclass nodes - these have no offspring. They act like a tag to the parent node indicating a particular flavour of that class. They are provided mainly for notational convenience, especially in the way that database entitities are defined.

4. atomic nodes - these are the offspring of lexical nodes. They represent actual values such as `Hotel Grand`, `Jazz`, `yes`, `27`, etc.

An example of a fully expanded user goal tree is shown in Fig. 4. This example is a simplified representation of a restaurant. The top level node represents an arbitrary entity. It has a subclass `venue` and corresponding subclass members `type`, `name`, and `location`. These members are generic for any kind of venue (e.g. restaurant, bar, hotel, etc). In this case, the type is a restaurant with restaurant-specific class members `food`, `music` and `decor`. The location is specified as a specific address and therefore has a `street` member. It could have been specified by some other means such as `nearto`, `gridref`, etc, and these would be alternate subclasses of location.

User goal trees are built using a set of rules which adhere to the syntax set out in Fig. 5[3]. As an example, the rules set out in Fig. 6 describe the restaurant goal described above. There are two basic forms of rules:

[3]Atomic names containing non-alphadigit characters must be enclosed in double quotes

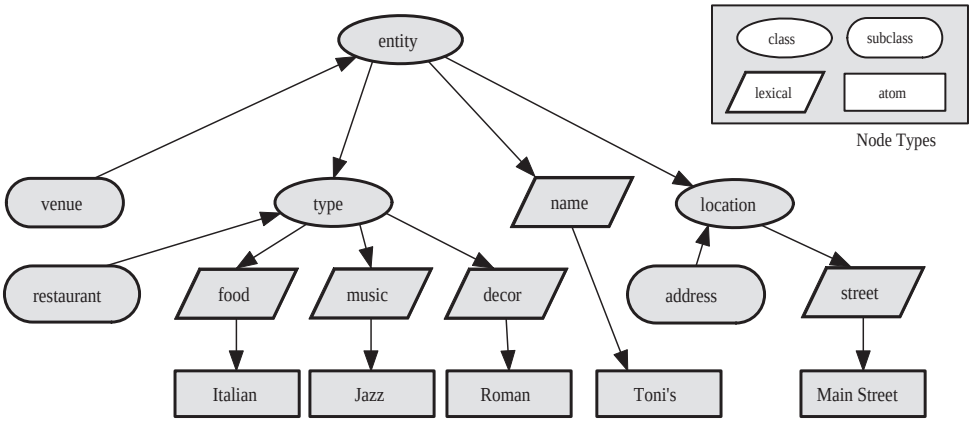

Figure 4: Example Fully Expanded User Goal Tree

class definition rules and lexical definition rules. The basic function of these should be clear from the table, however, some of the details require further explanation.

Firstly, the members of a class can have an optional "+" or "-" specifier indicating that the node is primarily *selectional* or *informational*, respectively. These markers are optional and only influence the selection of system responses. The plus specifier indicates that a value is normally required for that member in order to identify the requested entity. Conversely, the minus specifier indicates that the member will rarely be specified by the user to identify the entity but does contain information that the user may wish to know about once the entity has been selected. In the example rules, the food type is marked with a "+" since it is frequently specified by users in order to identify a suitable restaurant, whereas the `decor` is marked with a "-" since it is rarely specified by users when searching for an appropriate restaurant. It might, however, be required once a candidate restaurant has been located.

Secondly, note that in the left hand side of class definition rules, a simple name can be qualified using a dotted path notation. This is provided as a convenience to allow generic labels such as `name` to be used in different contexts, and then specific instances identified. In the example, the lexical definition for `name` is qualified by `venue` to distinguish it from other types of name.

Thirdly, a class definition rule can have simple equality constraints applied to its members. For example, in a travel booking system, a route might be specified as

```
route -> singleleg(fromplace,toplace) [toplace != fromplace];
```

In this case, any partition which instantiates the `fromplace` and `toplace` members with the same value will be marked as inconsistent and its belief will be set to zero.

Finally, all rules can have a probability assigned to them. Where no probability is given, then equal probability is assumed. These probabilities represent prior knowledge. In the example, the venue type is restaurant with probability 0.35. This would reflect the fact that in practice when users want to locate a venue, 35% of the time they require a restaurant. As explained in section 4.4, these prior probabilities are used to reallocate belief mass when a partition is split.

The ontology rules defined above describe the structure of the data. The data itself must be stored in a second file in the form of entity definitions, where each entity consists of a list of attribute value pairs. An example entity definition is shown in Fig. 7. Entity definitions must begin with an `id` attribute and should

```
ruleset   = ruledef";" { ruledef ";" } {dbasefile}
ruledef   = classdef | lexdef
classdef  = classinst "->" [subclass] [classbody] [cond] [prob]
classbody = "(" [opt] member { "," [opt] member } ")"
lexdef    = classinst "=" "(" atom[prob] {"|" atom[prob] ")"
prob      = "{" float "}"
cond      = "[" classinst op classinst "]"
opt       = "-" | "+"
classinst = name {"." name}
member    = name
subclass  = name
atom      = name
op        = "==" | "!="
dbasefile = "+" "filename"
```

Figure 5: Syntax of HIS Ontology Rules

```
entity     -> venue(type,name,location) {0.2};
type       -> restaurant(+food,music,-decor) {0.35}
location   -> addr(street) {0.8};
venue.name =  ("Toni's","Quick Bite", ....);
food       =  (Italian,Chinese,English, ...);
music      =  (Jazz,Pop,Folk, ...);
decor      =  (Traditional,Roman,ArtDeco,...
street     =  ("Main Street", "Market Square", ...);
```

Figure 6: Example of using Ontology Rules

normally include `name` and `type` attributes. All remaining attribute-value pairs are arbitrary but must be consistent with the rules. For example, all values must appear in at least one lexical definition[4].

The HIS system attempts to interpret attribute value pairs in a flexible way. For example, given the location rule in Fig. 6, an address could be specified by any of: `addr("Main Street")`, `location("Main Street")` or `street("Main Street")`. Note, however, that if there was also a rule such as

```
location -> nearto(street);
```

then the latter two forms would be ambiguous.

```
id("R23")
name("Toni's")
type("restaurant")
food("Italian")
addr("Main Street")
near("Cinema")
phone("2095252")
decor("Roman")
```

Figure 7: Example Database Entity Definition

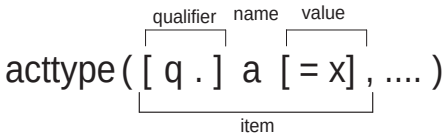

Figure 8: Structure of a Dialog Act

4.3 Dialog Acts

As shown in Fig 8, a dialog act consists of a type and a list of zero or more `name=value` pairs referred to as `items`. An item name refers to a node in a user goal tree, it can be a simple name or a qualified name where the qualifier is either the name of the parent node or the name of the parent's subclass, if any. There may be zero or many items in a single act, and the interpretation depends on the act type of which there are 15 in total. Note that this form of dialog act maps very naturally onto the DATE scheme, thus allowing easy integration into existing Talk systems.

The full set of acts supported by the HIS system is summarised in Table 1. The meaning of each act should be clear from the table, but the following amplifies a number of important points.

Firstly, the HIS system does not support multiple dialog acts in a single turn. Thus, for example, if

```
U: inform(food=Italian)
U: inform(music=Jazz)
```

is input to the system, it is interpreted as

```
U: inform(food=Italian) {0.5}
U: inform(music=Jazz)   {0.5}
```

i.e. the user said either that the food is Italian *or* that the music is Jazz with equal probability. To convey both pieces of information in a single turn, an `inform` act with two items must be used, i.e.

```
U: inform(food=Italian, music=Jazz)
```

In some cases, items are treated differently depending on their position in the item list. For example,

[4]Numbers are dealt with as a special case

```
S: confreq(type=restaurant,food)
```

is a request to confirm that the required type is restaurant and then request a value for food. If the response was

```
U: affirm(type=restaurant, food=Italian)
```

this would confirm the type and provide the required food information. In fact,

```
U: affirm(food=Italian)
```

would have the same effect since the sequence

```
S: confirm(type=restaurant)
U: affirm()
```

is identical to

```
S: confirm(type=restaurant)
U: affirm(type=restaurant)
```

If `negate` is used, however, the first item is always taken to be a correction thus the response

```
U: negate(food=Russian)
```

would be interpreted as “No, the food is not Italian, it is Russian”.

When an act is processed by the HIS system, its items are matched against the user goal tree. If a value is given, then an item can only match if there is an atomic leaf node with the same value and its parent (or the subclass of its parent) matches the name of the `name=value` pair. If no value is given then the name must match a node in the tree. If the name is qualified, then the qualifier must match the parent (or the subclass of the parent) of the matched node.

User dialog acts are presented to the system as lists of alternatives. Each alternative can have a probability attached to it. All acts without probabilities are assumed equally likely and assigned probabilities so as to make the total sum to one. Every input list must include a null dialog act with a non-zero probability. If no null act is included, the system inserts one.

4.4 Partitions and Partition Splitting

Section 4.2 explained how a single user goal is encoded in a branching tree structure. In fact, the HIS system maintains a forest of partially and fully-expanded trees. Each partially expanded tree represents a partition of equivalent user goal states. Each fully expanded tree is also a partition, but it is a singleton partition i.e., it encodes a single user goal state.

This forest of trees is stored in such a way that no partition is duplicated, and the sum of the probability of all partitions is always unity. As shown in Fig. 9(a), at system start up the user goal forest consists of a single node called `task`. This single partition p has belief $b(p) = 1$ and it represents all possible user goals. Since this node is built by default, all application rule sets must start with rules to expand this node. Thus, in practice, the rule set shown in Fig. 6 must be augmented by a rule such as:

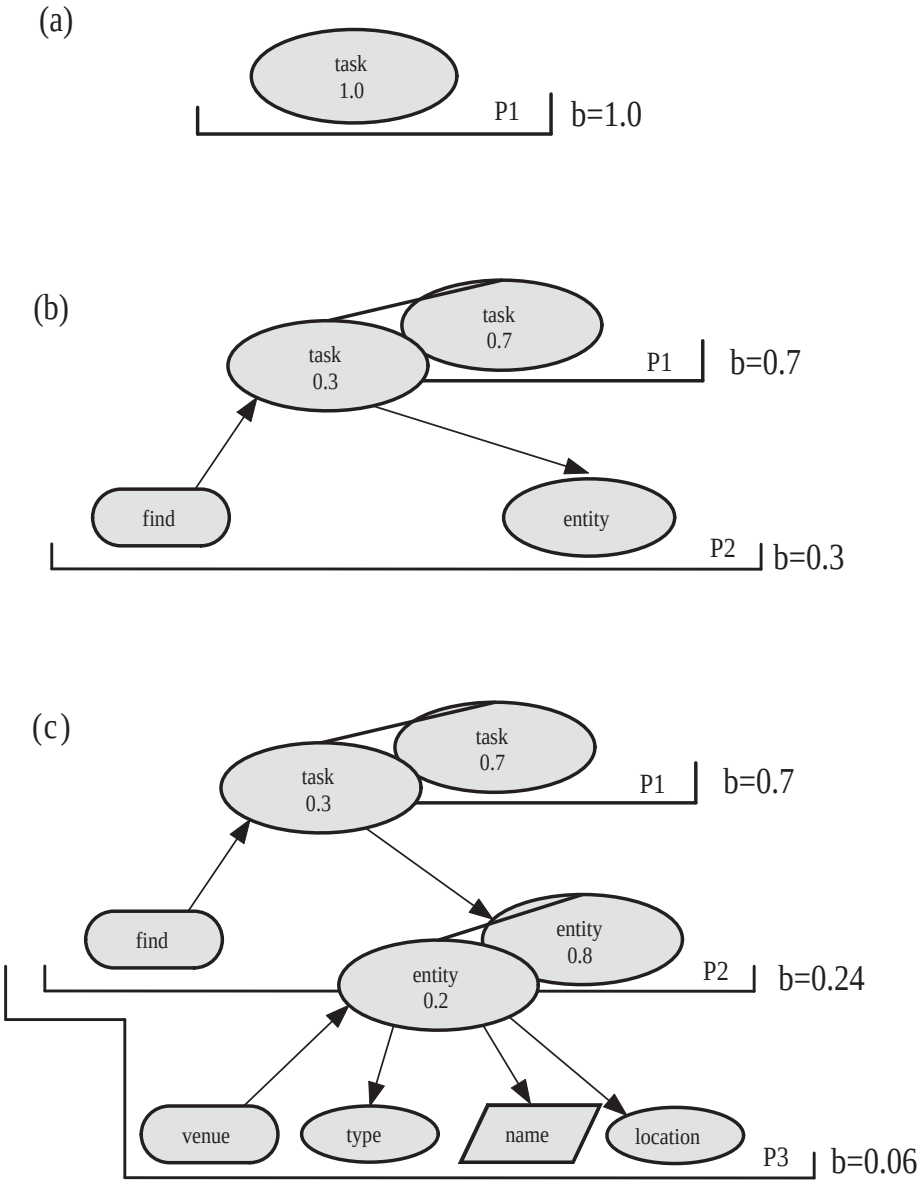

Figure 9: Example of Partition Splitting

Act	System	User	Description
hello()	√	√	start dialog
bye()	√	√	end dialog
inform(a=x,b=y,...)	√	√	give information a=x, b=y, ...
request(a,b,...)	√	√	request values for a,b, ...
confirm(a=x,b=y,..)	√	×	confirm a=x,b=y,..
confreq(a=x,..,c=z, d)	√	×	confirm a=x,..,c=z and request value of d
select(a=x,b=y)	√	×	select either a=x or b=y
affirm()	×	√	simple yes
affirm(a=x,b=y,...)	×	√	confirm a=x and give further info b=y, ...
negate()	×	√	simple no
negate(a=x,b=y,...)	×	√	no, a=x and give further info b=y, ...
repeat()	√	√	request to repeat last act
reqalts()	×	√	request alternative goal
reqalts(a=x,..)	×	√	request alt with new information
null()	√	√	null act - does nothing

Table 1: Supported Dialog Acts

```
task -> find(entity) {0.3};
```

which expresses the prior knowledge that 30% of the time, a user will wish to find something (e.g. a hotel, a restaurant etc). Fig. 9(b) shows what happens when this rule is applied. The `task` node is split into two parallel nodes and the probability mass is divided in proportion to the prior probability of applying the rule. The result is two partitions with beliefs $b = 0.7$ and $b = 0.3$ respectively. Suppose now that the rule for `entity` in Fig. 6 is applied, partition 2 is split to form a new partition and the belief mass is divided again. The result is as shown in Fig. 9(c). And so the process continues. The result in this case is three partitions which can be described via their leaf nodes as

```
P1: task {0.70}
P2: find(entity){0.24}
P3: find(venue(type,name,location)){0.06}
```

where the belief in each partition is shown in braces and always sums to one. Note that these prior beliefs give relatively high weight to unexpanded nodes because they represent the largest equivalence sets. However, once belief updating occurs, this situation is quickly reversed since the evidence typically supports only the more specific partitions.

The above explains how partitions are split but not when. In fact partition splitting is entirely on demand and it is driven by the items in the input user and system dialog acts. Referring back to Fig. 3, the first stage of the dialog cycle is to match the items of all of the input user acts and the previous system act against all of the existing partitions. Note that the act type is not relevant here since the goal is simply to expand the partitions sufficiently to match as many as possible of the input act items. Each item of each act is taken in turn and applied against each existing partition. If the item matches the partition, then the result is recorded and nothing further happens. If however the item does not match, then the ontology

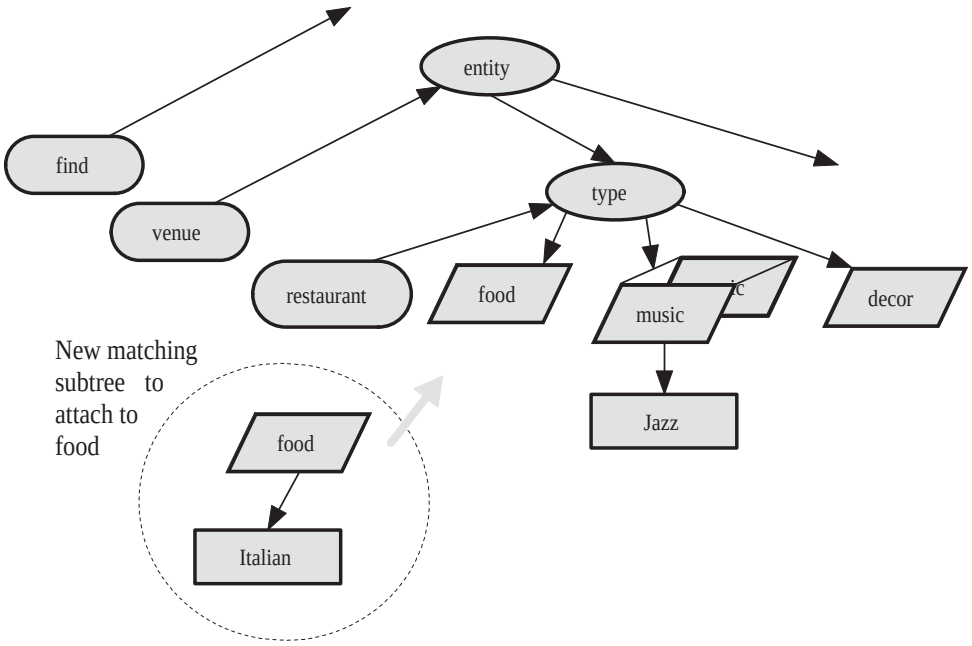

Figure 10: Splitting a Partition with a Shared Expansion Node

rules are scanned and the system tests to see whether the current partition could be extended sufficiently to allow the item to match. If it concludes that a match is possible, then the partition is extended and a match is recorded. For example, if the user goal forest was as shown in Fig. 9(b) at the point when the item (`music=Jazz`) was received, then the system would determine that a match could be achieved by first expanding the `entity` node using the first rule in Fig. 6. This node is referred to as the *expansion node*. The newly created offspring of the expansion node includes a `type` node and this can be expanded using the second rule in Fig. 6. Finally, expanding the lexical node `music` to derive the atomic node `Jazz` would allow the required match. Having determined that it is indeed possible to construct a *matching subtree* which if attached to the expansion node would support an item match, then that matching subtree is created.

The detailed implementation of this splitting process needs to consider a number of subtleties. Firstly, in order to ensure that all partitions are unique, a rule must be applied to a node only once. This is implemented by attaching to each expanded node, a reference to the rule used to expand it. It is then simple to check whether or not a rule has been applied before to that node, and if it has, the rule cannot be applied again. Secondly, when node expansion results in multiple levels of rule application, then new subtree nodes will be created with probability less than one. In each such case, a new parallel node must be created to hold the *unused* probability mass. Each new node created in this way creates a new partition. An example of this is shown in Fig. 9 where the expansion of partition `P1:task` to give partition `P3:find(venue(type,name,location))` results in an intermediate partition `P2:find(entity)` being created. In the further expansion needed to accommodate the item (`music=Jazz`), the expansion of the `type` node with probability 0.35 to `restaurant` would leave a parallel `type` node with probability 0.65 and this would form yet another partition.

Finally, as an act item is tested against successive partitions, there may be other partitions which have not

yet been examined but which share the same expansion node. Each of these as yet unexamined partitions, must be cloned and the expansion node replaced by the leaf nodes of the matching subtree. For example, in Fig. 10, there are two partitions

```
Px: find(venue(restaurant(food,music,decor)))
Py: find(venue(restaurant(food,music(Jazz),decor)))
```

If now the item `food=Italian` is matched against Py, then a new partition

```
Pyy: find(venue(restaurant(food(Italian),music(Jazz),decor)))
```

is created. However, the expansion node `food` is shared with Px, and hence a further partition

```
Pxx: find(venue(restaurant(food(Italian),music,decor)))
```

must also be created.

4.5 Constructing Hypotheses and the Dialog state

The previous subsections have explained how partitions are grown as a side effect of attempting to match dialog act items. Once all input items have been processed and all possible matches made, the next step is to construct a new set of updated beliefs for the current dialog turn. As indicated by Fig. 3, belief update is implemented by building an explicit list of hypotheses where each hypothesis corresponds to one possible combination of p', a'_u and s'_d in the left hand side of equation 20. At the start of each turn, each partition p has attached to it a list of possible dialog state records s_d where each combination $\{p, s_d\}$ corresponds to the final term in equation 20.

The dialog state records information about the dialog history which is relevant to the decision making process. In the current HIS system, this information consists of a count of the number of times a user goal node is referenced by the system and a count of the number of times a user goal node is referenced by the user. In future systems, this information will very likely be augmented. Indeed, the dialog state is an ideal place to embed heuristic knowledge into the system since the actual probability function $P(s'_d|p', a'_u, s_d, a_m)$ is deterministic and it is conditioned on the full system state space and the preceding dialog acts. From the programming perspective, this translates into a function which has all available infomation in its inputs and which returns 0 or 1.

During partition splitting, each derived partition inherits the full set of dialog state records from its parent partition. This generates the set $\{p', s_d\}$ with refined beliefs $P(p'|p)b(p, s_d)$. New hypotheses are then constructed by attaching all matching user acts a'_u to each refined partition. For each user act, all of the dialog state records attached to that partition are copied and attached to the user act. This generates all possible combinations of p', a'_u and s_d. The partitions s_d are updated and new beliefs $b(p', a'_u, s'_d) = b(h_k)$ are calculated according to equation 20.

Figure 11 illustrates hypothesis updating in more detail. In this example, the system had previously output `inform(music=Jazz)` and the user's response was either `request(food)` or `repeat()`. Previously there were two dialog states hypothesised for the given fragment of partition p' and after completing the turn, there are four distinct dialog states. This expansion occurs because the alternate user acts reference different elements in p' such that the user count for the `food` node is incremented in one case and not the other.

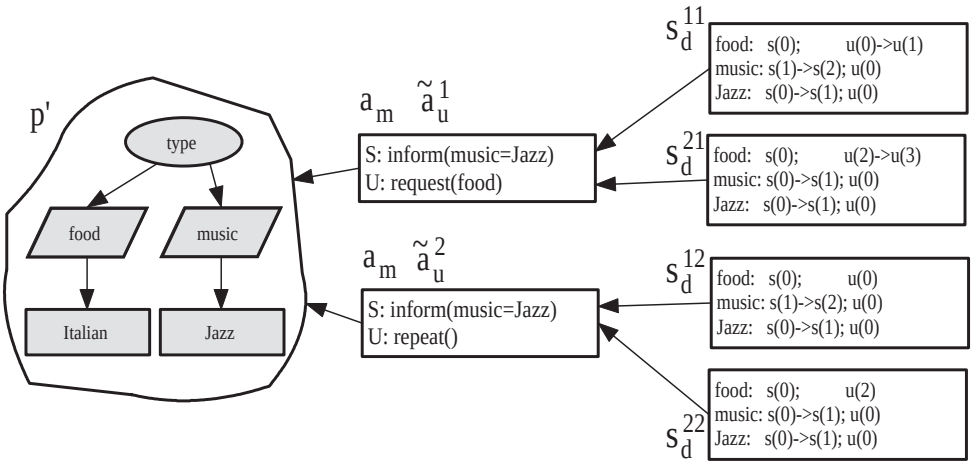

Figure 11: Example Hypotheses

5 A Greedy Theoretic Policy

As noted in the introduction and summarised in Figure 3, the current HIS system depends on a greedy theoretic policy i.e. a policy based on the computation of immediate rewards or utilities rather than in the case of a full POMDP where policies are based on expected future rewards.

The greedy policy is implemented as follows. Firstly, every hypothesis h_k is scanned and all possible system acts are proposed. Secondly, the candidate system acts are pooled and all duplicates removed. Next a utility $U(a_m^i|h_k)$ is computed for each candidate system act a_m^i given hypothesis h_k. The utility of each candidate system act is then averaged across all hypotheses (i.e. beliefs), and the act with the highest expected utility is selected for output, i.e.

$$\tilde{a}_m^* = \operatorname*{argmax}_i \left\{ \sum_k b(h_k) U(a_m^i|h_k) \right\} \tag{27}$$

5.1 Generating Candidate System Acts

The generation of candidate system acts is rule based. Firstly, the terminals of each hypothesis are used as search keys into the database. If the number of matching entities is exactly one[5], then the entity is bound to that hypothesis. If the number of matching entities is zero then the hypothesis is marked as *overspecified* and never considered again. For each hypothesis, the following generation rules are then applied:

- if the partition is bound to an entity, then for every attribute value pair $a = v$ in that entity which has never been mentioned before, propose the act *inform(a=v)*.
- if the partition is bound to an entity and all terminal values have been instantiated and mentioned to the user at least once, propose the act *bye()*.

[5]In future versions of the system, this will be increased to a small number of candidate entities to provide the user with a choice.

- if the partition is not bound to any entity, scan leaves and if any non-atomic leaf *a* is marked as a select key (i.e. it was marked with a "+" in the ontology rules), then propose the act *request(a)*.
- if the partition is not bound to any entity and there are no select keys, then for every non-atomic leaf *a* propose the act *request(a)*.
- if the partition is bound to an entity, then for every atomic leaf value *v* with a user count of just 1 and with parent or subclass *a*, propose the act *confirm(a=v)*.

After the above single item candidate system acts have been generated, the pool is rescanned and multiple item acts are generated. For example, if two confirm acts refer to differing values v_1 and v_2 of the same attribute *a*, then `select(a=v1,a=v2)` is generated.

5.2 Computing Utilities

Given a specific hypothesis, the utility of a candidate system act with respect to that hypothesis is computed as follows. Firstly, four heuristic metrics are computed:

risk - this measure is based on the ratio of the number of confirmed terminals to the total number of instantiated terminals where a terminal is judged to have been confirmed if its user count is greater than 1.

progress - this measure is based on the distance to go in terms of terminal instantiation before the partition will bind with a unique entity in the database.

relevance - this attempts to measure the relevance of the proposed system act given the hypothesised user act and the partition information. It is act dependent and entirely heuristic.

continuity - this measure is a combination of two sub-measures: the degree to which the candidate system act is in focus with respect to the previous user act and the bigram probability of the system act type given the previous user act type. The first sub-measure depends only on the items in the system act and it is computed by counting the common ancestors in the partition tree of the system act items and the user act items. The second sub-measure depends only on the act types.

Once these four measures have been computed, the utility of a candidate system act a_m^i given the hypothesis h_k is calculated by:

$$U(a_m^i|h_k) = \alpha_{risk}[t] * \text{risk} + \alpha_{prog}[t] * \text{prog} + \alpha_{relev}[t] * \text{relev} + \alpha_{cont}[t] * \text{cont} \quad (28)$$

where t is the type of a_m^i. α is an array of act type dependent weights which allow different emphasis to be placed on different types of system act. For example, α_{risk} is negative for confirm acts and positive for inform acts, thus encouraging caution when the risk is high.

6 Conclusions and Further Work

This report has outlined a new Bayes Net framework called the Hidden Information State (HIS) model for designing and implementing spoken dialog systems. The model is based on the SDS-POMDP but it

avoids the usual computational issues associated with POMDPs by partitioning the space of user goals into a small number of equivalence classes. Probabilistic context-free ontology rules are used to describe the iterative splitting of partitions to eventually form unique goal states. By computing beliefs on partitions rather than the underlying states, belief monitoring remains tractable even for complex real-world systems.

This initial version of the HIS system relies on several hand-crafted probability tables and a greedy policy based on somewhat *ad hoc* metrics. The next phase of the development will include training these tables from data and performing comparitive evaluations of the system using a hand-crafted system as the baseline.

In addition to the need for proper training, there is also an outstanding problem relating to priors. In the current system, priors are represented by the probabilities of context-free rewrite rules. These rules are context independent and take no account of the very limited number of actual database entities available to match the fully expanded user goal trees. The net effect is that the model underestimates prior probabilities, especially the probabilities of singleton partitions. This problem is currently mitigated by flooring expansion rule probabilities but this is a very crude solution and it needs improving.

The current system is also limited to handling a single static user goal. Further work is needed to expand the framework to support changing user goals.

Overall the HIS system is believed to represent a major step in solving the problem of scaling-up SDS-POMDPs to handle real world applications. The next major step is to find methods of constructing efficient policies which incorporate planning and this is the topic of future research. In the meantime, armed with a utility-based greedy theoretic planning algorithm, the inherent robustness of the HIS model to understanding errors should enable it to compete with and perhaps exceed the performance of existing dialog systems, even without the benefit of a globally optimised POMDP policy.

References

[1] S Larsson and D Traum. Information State and Dialogue Management in the TRINDI Dialogue Move Engine Toolkit. *Natural Language Engineering*, pages 323–340, 2000.

[2] SJ Young. Talking to Machines (Statistically Speaking). In *Int Conf Spoken Language Processing*, Denver, Colorado, 2002.

[3] E Levin, R Pieraccini, and W Eckert. A Stochastic Model of Human-Machine Interaction for Learning Dialog Strategies. *IEEE Trans Speech and Audio Processing*, 8(1):11–23, 2000.

[4] LP Kaelbling, ML Littman, and AR Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101:99–134, 1998.

[5] JD Williams, P Poupart, and SJ Young. Factored Partially Observable Markov Decision Processes for Dialogue Management. In *4th Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Edinburgh, 2005.

[6] JD Williams, P Poupart, and SJ Young. Partially Observable Markov Decision Processes with Continuous Observations for Dialogue Management. In *6th SIGdial Workshop on DISCOURSE and DIALOGUE*, Lisbon, 2005.

[7] ML Littman. The Witness Algorithm: solving partially observable Markov decision processes. Technical report, Brown University, December 1994 1994.

[8] T Paek and E Horvitz. Conversation as Action Under Uncertainty. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 455–464, Stanford, CA, 2000. Morgan Kaufmann.

[9] B Zhang, Q Cai, J Mao, and B Guo. Planning and Acting under Uncertainty: A New Model for Spoken Dialogue System. In *Proc 17th Conf on Uncertainty in AI*, Seattle, 2001.

[10] MTJ Spaan and N Vlassis. Perseus: randomized point-based value iteration for POMDPs. Technical report, Universiteit van Amsterdam, 2004.

[11] JD Williams and SJ Young. Scaling up POMDPs for Dialogue Management: the Summary POMDP Method. In *IEEE workshop on Automatic Speech Recognition and Understanding (ASRU2005)*, Puerto Rico, 2005.

[12] S Singh, DJ Litman, M Kearns, and M Walker. Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System. *J Artificial Intelligence Research*, 16:105–133, 2002.

[13] K Scheffler and SJ Young. Automatic Learning of Dialogue Strategy using Dialogue Simulation and Reinforcement Learning. In *HLT 2002*, San Diego, USA, 2002.

[14] AW Moore and CG Atkeson. The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces. In SJ Hanson, JD Cowan, and CL Gi, editors, *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 1994.

[15] AK McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, 1995.

[16] WTB Uther and MM Veluso. Tree Based Discretization for Continuous State Space Reinforcement Learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–775, 1998.

[17] MJ Kochenderfer and G Hayes. Adaptive Partitioning of State Spaces using Decision Graphs for Real-Time Modeling and Planning. In *Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains, IJCAI-05*, Edinburgh, 2005.

[18] R Jaulmes, J Pineau, and D Precup. Active Learning in Partially Observable Markov Decision Processes. In *European Conference on Machine Learning (ECML)*, Porto, Portugal, 2005.

Part II
Software Implementation

7 HIS Prototype Software Structure

7.1 Overall Structure

The prototype HIS system is implemented as a C++ class library. As illustrated by the dependency diagram in Fig. 12, the software consists of six main modules:

DMan the `DMan` class provides the primary interface to the dialog manager. Its principal function is to accept each incoming list of user dialog acts, apply each act to the user goal tree and then bind the acts to matching partitions.

DialogState the `DialogState` class maintains a list of dialog hypotheses and a list of candidate system acts. Each hypothesis is represented by an instance of the `Hypothesis` class and each candidate is represented by an instance of the `CandidateSysAct` class.

UserState this is the most complex module in the library. It provides the top level `UserState` class, the `Partition` class for representing each partition of the user state space, and the `TreeNode` class for building the actual user goal trees.

Rules this module defines two main classes: `RuleTable` and `Database`. These data structures provide the internal representations of the application definition rules and database files.

Lex this is a simple lexical scanner used for parsing both the external definition files, and text string representations of dialog acts.

7.2 A Simple Application Example

To build an application using the HIS manager, the application ontology rules and the database entities must be defined. Examples of a simple application in the tourist domain are given in Fig. 13 and Fig. 14[6]

The `DMan` interface class provides a constructor and four basic methods for driving the dialog. The name of the rules file is given to the `DMan` constructor as an argument, the corresponding database file name(s) are listed in the rules file. Building the HIS dialog manager into an application is very simple. As illustrated in Fig. 15, a DMan instance is created, and then the four interface methods `StartTurn`, `DOut`, `DIn`, `EndTurn`, are repeatedly called in sequence.

Although the HIS system is primarily a class library, the prototype software distribution includes a simple terminal input/output program called `TDMan` which allows the “user” to type in user acts in response to system acts.

An example session using this program in the SACTI tourist information domain is shown in Fig. 16. This is a straightforward example in which the user requests a hotel. The system asks for a location and a price,

[6]These examples are simplified versions of the SACTI tourist domain.

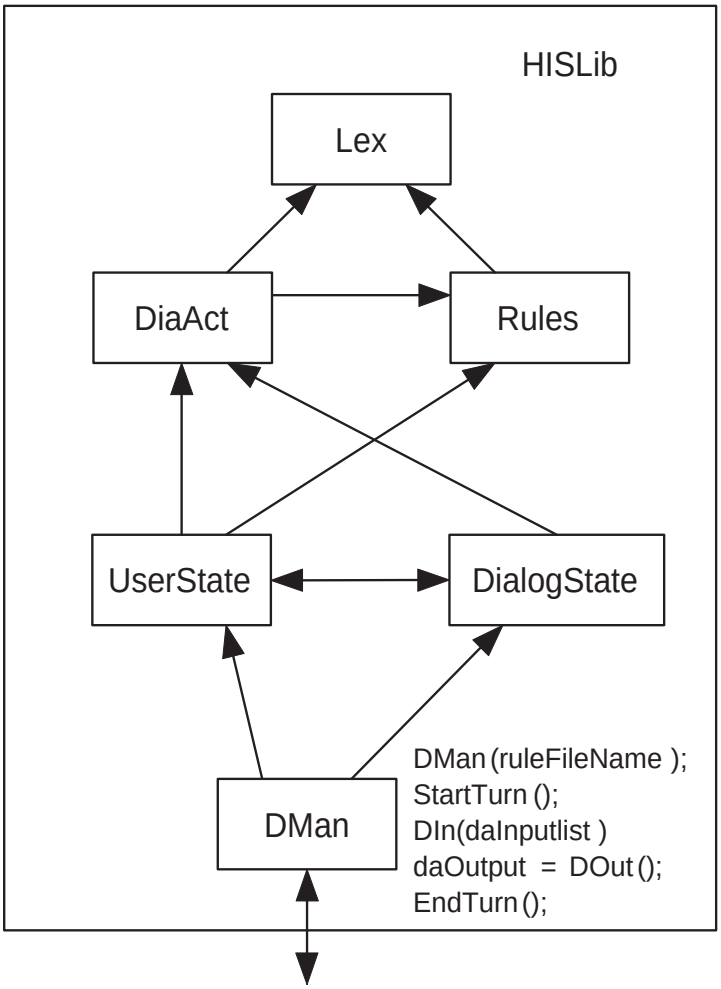

Figure 12: Overall Software Structure

and based on this suggests the “Hotel President”. The user requests the phone number and address and then closes the dialog. Fig. 17 shows a second example where the user requests a restaurant. This time when the system requests a location, the user ignores this and requests a specific food type. A recogniser ambiguity is simulated by entering two values for the food type. The system identifies two similar states with similar beliefs and it outputs a select act to resolve the ambiguity.

The HIS software modules have extensive trace output capability which is documented in the headers of the implementation files. Tracing is enabled by setting bits in a TRACE configuration variable as in HTK and ATK [7]. As an example, Fig. 18 shows the active partitions and the candidate system act list at turn 3 of the restaurant dialog above.

[7] See htk.eng.cam.ac.uk

```
# HIS Rules for simple tourist information system

# define tasks, just one here but more in practice
task -> find (entity){0.4};

# define the entities to find
entity -> +venue(-name, +type, +loc, -addr, -phone);
type -> +hotel(+pricerange, -ensuite, -roomsize, -withBreakfast){0.4};
type -> +restaurant(+food,  +pricerange, -pricemeal, -music){0.3};
type -> +bar(+drink, -music);

# define generic entitities
addr -> (street);
loc -> xing(street) {0.3};   loc -> near(street) {0.7};
ensuite -> (boolean);        withBreakfast -> (boolean);
pricemeal -> (number);       phone -> (number);

size = (large | medium | small );
boolean = (yes|no);          number =(0);

# enumerate all the atomic types

venue.name = ("Hotel Primus" | "Hotel President" | "Royal Hotel" |
             "Bochka" | "Siberian Tiger" | .... );
music = ("Bolchoi musicians"|"soft jazz" | "rock" | "classical" );
street = ( "Alexander Street" | "Castle Loop" | "North Road" |
         | "Main Street" | "Fountain Road" | "West Loop" | ... );
food = ( snacks  | sandwiches | Russian | French  | ... );
drink = ( beer | "Mexican beer" | wine | stouts | cocktails  );
pricerange = ( expensive | moderate | inexpensive | cheap );

# define dbase data files - must be in same directory as rule files
+"dbase.txt"
```

Figure 13: Example Application Ontology Rules

```
id("H1")
name("Hotel Primus")
type("hotel")
addr("Alexander Street")
xing("West Loop")
near("Main Square")
phone("2094227")
pricerange("moderate")
ensuite("yes")
roomsize("large")

id("R2")
name("Siberian Tiger")
type("restaurant")
addr("West Loop")
near("Fountain Road")
phone("7095926")
pricemeal("38")
pricerange("expensive")
food("Russian")
music("Bolchoi musicians")

...etc
```

Figure 14: Example Database Entity Definitions

```
// global variables
DiaActPtrList dalist;    // list of input dialog acts from user
DiaActPtr    sysact;     // system dialog act for output to user
DMan         *dm;     // HIS dialog manager

// instantiate a HIS dialog manager using the application
// rules and dbase stored in application.txt
DMan *dm = new DMan("application.txt");

// loop endlessly send input dialog acts to dm, and
// recieving system acts to output.  OuputSystemAct
// and GetDialogActList represent the rest of the
// dialog system external to the HIS manager.

do {
   // start a new turn
   dm->StartTurn();

   // output prompt to user.  Eg convert the
   // dialog act to words and send to synthesiser
   sysact = dm->DOut();
   OutputSystemAct(sysact);

   // input list of dialog acts from user.  Eg
   // extract N-best list from recogniser, and
   // parse to extract
   dalist = GetDialogActList();
   dm->DIn(dalist);

   // signal end of turn
   dm->EndTurn();
}
```

Figure 15: Application Program Structure

```
Sys> hello()
DA1> request(hotel)

Sys> confirm(type.hotel)
DA1> affirm(near=Museum)

Sys> confirm(near=Museum)
DA1> affirm()

Sys> request(hotel.pricerange)
DA1> inform(=moderate)

Sys> confirm(pricerange=moderate)
DA1> affirm()

Sys> inform(name="Hotel President")
DA1> request(phone)

Sys> inform(phone=7192277,name="Hotel President")
DA1> request(addr)

Sys> inform(addr="Castle Loop",name="Hotel President")
DA1> bye()
```

Figure 16: Example Dialog 1: Simple Hotel Request

```
Sys> hello()
DA1> request(restaurant)

Sys> confirm(type.restaurant)
DA1> affirm()

Sys> request(venue.loc)
DA1> inform(food=Indian){0.6}
DA2> inform(food=Russian){0.4}

Sys> select(food=Indian,food=Russian)
DA1> inform(food=Indian)

Sys> inform(restaurant.name="Saint Petersburg")
DA1> bye()
```

Figure 17: Example Dialog 2: Restaurant Request

```
Turn: 3: System Act = request(venue.loc) P(Au|O)=1.00

Partition[5]: unique  belief=2.00e-001 treeprob=0.024 parent=2

|    ---------------------------------task
|   /                                    \
| find     ----------------------------entity-----------------------------
|         /                           \                    \    \     \     \
|       venue        ------------type---------------     name  loc  addr  phone
|                   /         \       \      \      \
|              restaurant   food   prange  pmeal  music
|                             |
|                           Russian
|

Partition[4]: unique  belief=2.00e-001 treeprob=0.024 parent=2

|    ---------------------------------task
|   /                                    \
| find     ----------------------------entity-----------------------------
|         /                           \                    \    \     \     \
|       venue        ------------type---------------     name  loc  addr  phone
|                   /         \       \      \      \
|              restaurant   food   prange  pmeal  music
|                             |
|                           Indian
|

Partition[2]: generic  belief=6.00e-001 treeprob=0.072 parent=0

|    ---------------------------------task
|   /                                    \
| find     ----------------------------entity-----------------------------
|         /                           \                    \    \     \     \
|       venue        ------------type---------------     name  loc  addr  phone
|                   /         \       \      \      \
|              restaurant   food   prange  pmeal  music
|

Partition[3]: generic  belief=5.00e-003 treeprob=0.280 parent=0

|    ---------------task
|   /                   \
| find     -----------entity-----------
|         /      \     \    \     \     \
|       venue  type  name  loc  addr  phone
|

Partition[1]: generic  belief=5.00e-003 treeprob=0.600 parent=0

| task
|

5 partitions: 0 init, 3 gen, 0 grp, 2 uniq, 0 ospec, 0 incon
Like Ratio = 1.00; Tree prob = 1.000
=============================================================================================================

Candidate System Acts
  select(food=Indian,food=Russian) Util = 0.172
  request(restaurant.prange) Util = -0.032
  request(restaurant) Util = -0.032
  request(venue.loc) Util = -0.065
  request(venue) Util = -0.065
  request(restaurant.food) Util = -0.131
  request(venue.type) Util = -0.230
  confirm(food=Indian) Util = -1.331
  inform(name=Saint Petersburg) Util = -1.692  Entity = restaurant:Saint Petersburg
  inform(pmeal=20,name=Saint Petersburg) Util = -1.797  Entity = restaurant:Saint Petersburg
  inform(prange=moderate,name=Saint Petersburg) Util = -1.797  Entity = restaurant:Saint Petersburg
  inform(addr=Park Road,name=Saint Petersburg) Util = -1.817  Entity = restaurant:Saint Petersburg
  inform(phone=7812311,name=Saint Petersburg) Util = -1.817  Entity = restaurant:Saint Petersburg
  inform(food=Indian,name=Saint Petersburg) Util = -2.276  Entity = restaurant:Saint Petersburg
  confirm(food=Russian) Util = -2.532
  inform(name=Siberian Tiger) Util = -2.772  Entity = restaurant:Siberian Tiger
  inform(music=Bolchoi musicians,name=Siberian Tiger) Util = -2.842  Entity = restaurant:Siberian Tiger
  inform(pmeal=38,name=Siberian Tiger) Util = -2.842  Entity = restaurant:Siberian Tiger
  inform(prange=expensive,name=Siberian Tiger) Util = -2.842  Entity = restaurant:Siberian Tiger
  inform(addr=West Loop,name=Siberian Tiger) Util = -2.855  Entity = restaurant:Siberian Tiger
  inform(phone=7095926,name=Siberian Tiger) Util = -2.855  Entity = restaurant:Siberian Tiger
  inform(food=Russian,name=Siberian Tiger) Util = -3.162  Entity = restaurant:Siberian Tiger
  inform(near=Fountain,name=Siberian Tiger) Util = -4.932  Entity = restaurant:Siberian Tiger
  inform(near=Park,name=Saint Petersburg) Util = -4.932  Entity = restaurant:Saint Petersburg
```

Figure 18: Example of System Tracing

Appendix - contents of the D4.3 deliverable CD

The prototype deliverable for D4.3 consists of a C++ class library and a simple demonstration of its use. The CD contains:

1. a PDF version of this report
2. a folder called `include` containing all the HIS header files
3. a folder called `windows` containing a compiled HIS class library `HISLib.lib`
4. a folder called `linux` containing a compiled HIS class library `HISLib.a`
5. a folder called `demo` containing a simple driver program for the HIS library called `TDMan.cpp` consisting of the executables `demo.exe` for Windows and `demo` for Linux, a configuration file called `cfg`, a set of example ontology rules called `rules.txt` and a small database called `dbase.txt`.

To run the demo, simply type

```
demo -C cfg rules.txt
```

To run one of the two canned dialogs listed above, type

```
demo -C cfg -n 1 rules.txt
```

or

```
demo -C cfg -n 2 rules.txt
```

To change the trace output, modify the flag bits in `cfg`.

To compile a program, include the header files in `include` and link with the appropriate version of `HISLib`.