Learning Task Models for Collaborative Discourse

Andrew Garland and **Neal Lesh** and **Candace Sidner**
Mitsubishi Electric Research Laboratories
{garland,lesh,sidner}@merl.com

Abstract

Combining general principles about collaboration with a task model for a specific environment allows an agent to adapt its utterences based upon the history of interactions with the user. However, developing models that can be used by a collaborative agent is a significant engineering challenge. Learning techniques that infer an accurate model for a given task from annotated examples can lessen this burden considerably. However, there are is still a noticeable disparity between an accurate model and a model that results in dialogs that a human user is comfortable with.

1 Background

In the Collagen (Rich et al., 2001) architecture, a collaborative interface agent engages in dialogs with a user so that, together, they can jointly achieve tasks. Adaption in these dialogs arises from the agent's internal model of the conversation and activity thus far (the so-called dialogue context) which is guided by a declarative model of knowledge about tasks. Thus, dialog adaptation occurs as a byproduct of a general theory of collaboration.

Collagen implements the SharedPlans theory of collaborative discourse (Grosz and Sidner, 1990), in which possible utterances of the agent are determined by a general-purpose algorithm for discourse interpretation (Lochbaum, 1998). Because the agent uses a declarative model of actions and tasks, this approach gives rise to the notorious *knowledge acquisition bottleneck*: developing an effective model is a significant obstacle to applying general AI methods to a domain. Our goal is to support a domain expert in the construction of such models with machine learning techniques.

This work presents techniques to automatically acquire *task models*, i.e., declarative representations of how to decompose and accomplish goals and subgoals. A distinguishing characteristic of our techniques is that they focus on one of the most difficult aspects of learning task models: deciding how to divide tasks into subtasks, which involves choosing the best abstractions for intermediate goals.

The choice of intermediate goals is especially important for collaborative agents because the agent must be able to discuss how to accomplish tasks in a way that is intuitive to the user. Our approach to acquiring task models is based on the conjecture that it is more difficult for people to deal with abstractions in the task model than to generate and discuss examples of how to accomplish tasks. In other words, we are designing a *programming by demonstration* (Cypher, 1994) system in which a domain expert performs some task by executing primitive actions and then reviews and annotates a log of their actions.

A task model influences dialogs but does not control them. The model allows an agent and the user to maintain shared beliefs about their efforts. Based on these mutual beliefs and the agent's level of initiative, the agent might make a variety of utterances. Thus, dialogs automatically adapt to reflect the history of interactions between the human the user in a collaboration. A limitation of a single unchanging model is that it does not allow for on-line adaptation: the same sequence of user actions in a future collaboration will produce the same dialogs. An area for future work is to allow for changing models over time.

The next section of the paper will talk about how task models are inferred from *partially-annotated* examples of task-solving behavior. The different types of partial annotations are described, and an example that illustrates the type of inferences made by the learning algorithm is presented. Empirical results are included that quantify how the different types of annotations influence the number of examples that need to be provided by the domain expert.

A discussion section of the paper describes how the models that are learned by these techniques are inadequate for a collaborative agent, despite the fact that they provide accurate domain models. An obvious deficiency is that automatically generated names for various pieces of the model will not be meaningful to a human. A less obvious one is that some intermediate goals do not require parameters for correctness, but the resulting conversations flow less naturally.

2 Learning Task Models

Before going on to describe the techniques that learn task models, we will describe the models themselves. Describing how these models can be used by a collaborative agent to generate utterances is beyond the scope of this paper. The interested reader is directed to Rich et al. (2001) for an overview; Lesh et al. (1999) discusses the role of plan recognition in extending the basic discourse interpretation algorithm (Lochbaum, 1998); and Lesh et al. (2001) discusses how possible actions and utterances are selected from an agenda of possibilities, given the current discourse state.

A task model is composed of actions and recipes. Actions are either primitive actions, which can be executed directly, or non-primitive actions (also called "intermediate goals" or "abstract actions"), which are achieved indirectly by achieving other actions. Each action has a type; each action type is associated with a set of parameters, but does not have an explicit representation of causal knowledge for its preconditions and effects.

Recipes are methods for decomposing non-primitive actions into subgoals. There may be several different recipes for achieving a single action. Each recipe describes a set of steps that can be performed to achieve a non-primitive action. A recipe also contains constraints that impose temporal partial orderings on its steps, as well as other logical relations among their parameters. For the purposes of this paper, however, we will consider only equality relationships. Equalities between a parameter of a step and a parameter of the action being achieved by the recipe are called bindings, but are otherwise indistinguishable from constraints. All steps are assumed to be required unless they are labelled as optional.

nonprimitive act PreparePasta
  parameter Pasta pasta

recipe PastaRecipe **achieves** PreparePasta
  steps Boil boil
    CookPasta cook
    optional GetPasta get
  bindings **achieves**.pasta = make.pasta
  constraints get.item = cook.pasta
    boil.liquid = cook.water
    boil **precedes** make
    get **precedes** make

primitive act GetPasta
  parameter Item item

Figure 1: Collagen representations from a simple cooking domain.

Figure 1 contains samples of these representations. Keywords for the task model language are in bold. Parameters and steps have a name as well as a type in order to allow for unambiguous references in bindings and constraints. A task model in the form of Figure 1 is the desired output of learning; acting from such a model will allow an agent to automatically adapt its utterances to match the specific dialog context.

Annotation language Informally, the input to the learning algorithm is a series of demonstrations; each one shows one correct way to perform a task and indicates that examples similar to that one are also correct. More precisely, an annotated example e is a five-tuple: $\langle \hat{e}, \mathcal{S}, \mathit{optional}, \mathit{unordered}, \mathit{unequal} \rangle$:

$\hat{e}$ is the temporally ordered list of primitive actions $[p_1, \ldots, p_k]$ that constitute the unannotated example demonstrated by the expert.

$\mathcal{S}$ is a segment; a segment is a pair $\langle \mathit{segmentType}, [s_1, \ldots, s_n] \rangle$. Each s_i, called a *segment element* or element for short, is either a primitive action or is a segment. Grouping elements together means that, as a unit, they logically form one occurrence of an intermediate goal of type *segmentType*.

optional is a partial mapping from segment elements to boolean values. If the mapping is defined and is true, the expert is specifying that removing that element from the example would constitute another correct example from the domain.

unordered is a partial mapping from pairs of elements in the same segment to boolean values. If the mapping is defined and is true, the expert is specifying that switching the order of appearance of the pair of elements would constitute another correct example from the domain.

unequal is a partial mapping from pairs of action parameters to boolean values. If the mapping is defined and is true, the expert is specifying that another correct example with the same segmentation exists wherein these two parameters do not have the same value.

Figure 2 depicts the amount and type of information that a domain user would have to add when annotating a raw usage log. In this example, the information is textual, but, in practice, a graphical tool would likely be preferred. The italicized text is the raw usage log: each primitive action is subscripted so that different instances of the same act type can be distinguished. The arguments of each primitive action are specific domain items.

The non-italicized text in Figure 2 show the annotations of the domain expert. In this figure, the elements of a segment are identified visually by the

segment of type MakeMeal_1	
segment of type PreparePasta_1	unordered PrepareSauce_1, precedes ServeDinner_6
$Boil_1(water_9)$	unordered GetPasta_2, precedes CookPasta_3
$GetPasta_2(linguini_4)$	precedes CookPasta_3, optional
$CookPasta_3(linguini_4, water_9)$	Boil_1.liquid = CookPasta_3.water, GetPasta_2.item = CookPasta_3.pasta
segment of type PrepareSauce_1	precedes ServeDinner_6
$Boil_4(water_9)$	precedes MakeSauce_5, optional, Boil_1.liquid $\neq$ Boil_4.liquid
$MakeSauce_5(bolognese_7)$	
$ServeDinner_6(kitchen_5)$	

Figure 2: Annotated usage log. The raw usage data is in italics, all of the other text is annotations. Note that this text-based annotation language reflects the underlying representation language (in Figure 1), but is not identical. Annotations in the left half of the figure must be provided the domain expert; while those on the right-hand side are not required by the learning techniques, they speed learning.

level of indentation. Also, the mappings described above are presented in a less formal manner; instead of defining $unordered(a, b) = \text{true}$, action a is annotated with "unordered b". If a is annotated with "precedes b", this means $unordered(a, b) = \text{false}$.

If one can assume that a domain expert will be meticulous and always mark certain types of annotations, default reasoning can lighten the expert's load. For example, in Figure 2, many steps are not marked as optional. Although one might intuitively read this to mean that the unmarked steps are required, it might just be the case that the expert is not sure if those steps are required or optional. While it is tempting to rely on defaults, problems arise if experts are imperfect annotators, so our learning techniques are conservative and do not make inferences from the absence of annotations.

It should be clear from even this simple example that fully annotating examples could be quite burdensome.

2.1 Learning algorithm

Figure 3 contains pseudo code for our task model learning algorithm, which requires polynomial time. In this figure, $\bar{\mathcal{E}}$ refers to the set of annotated examples from which a model is being derived and m_i represents a task model.

A fundamental search problem, which we refer to as alignment, faced by any task learning algorithm is to determine which primitive actions, possibly in different examples, correspond to the same recipe step. Additionally, an algorithm for learning hierarchical task models must also match segments to recipes. This is a fundamental problem because a learning algorithm needs to identify segments with recipes and segment elements with recipe steps in order to update the model.

LEARNMODEL $(\bar{\mathcal{E}}) \equiv$
  $m_0 \leftarrow$ ALIGNMENT$(\bar{\mathcal{E}})$
  $m_1 \leftarrow$ INDUCEOPTIONAL$(m_0, \bar{\mathcal{E}})$
  $m_2 \leftarrow$ INDUCEORDERING$(m_1, \bar{\mathcal{E}})$
  return INDUCEPROPAGATORS$(m_2, \bar{\mathcal{E}})$

Figure 3: Pseudo code to learn a task model

Suppose, for example, that a human expert indicates that $[a, b, c]$ and $[c, b, a]$ both achieve goal Z. The alignment question, here, is whether to learn one or two recipes for Z. Without an assumption or heuristic, there is no justification to learn only one recipe. But if we never combine multiple examples into one recipe, we can not perform any useful generalization.

Alignment is intractable in the absence of assumptions about the domains being studied (i.e., restricting the class of models being learned). However, we render the alignment problem tractable by making the following fairly benign assumptions that restrict the class of task models our algorithm will learn:

Disjoint steps assumption: for any two recipes that achieve an action of the same type, the sets of the types of their required steps will not be in a subset / superset relation.

Step type assumption: if any recipe contains multiple steps of the same type, they will be totally ordered and only the last might be optional.

If alignment is correct, the learning algorithm will be able to make logically correct inferences from the annotated examples. For example, if the sequence $[a, b, c]$ is correct and b is annotated as optional, then we know $[a, c]$ is also a correct example. We can also generalize from the annotated examples based on assumptions about the target model to be learned. For example, if the learner is told $[a, b, c]$ and $[c, b, a]$ are both correct and the target model

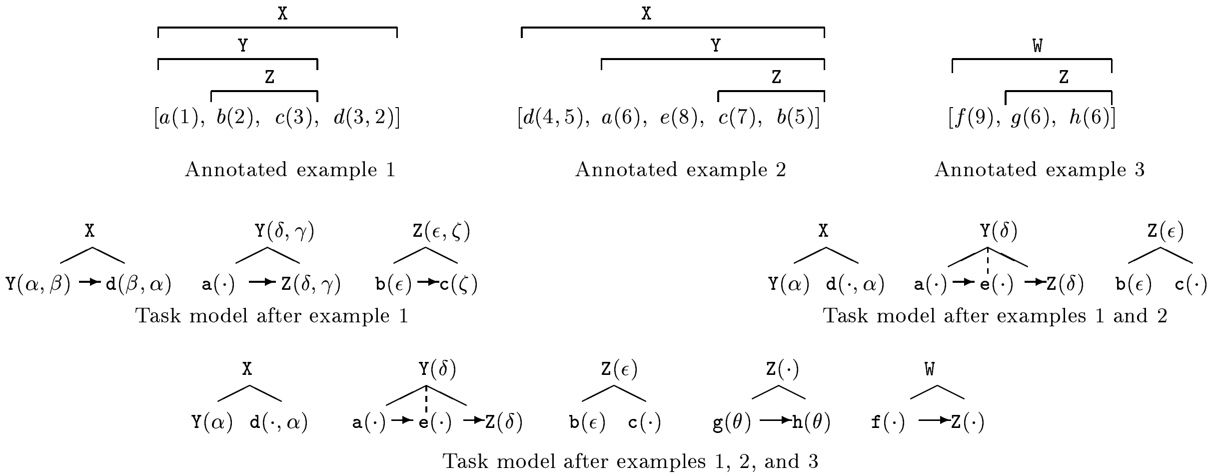

Figure 4: Graphical representations for annotated examples and learned task models for an abstract domain. The unannotated examples are sequences of instances of primitive actions, shown in lower-case italic letters, such as $[a(1), b(2), c(3), d(4)]$. The action $a(2)$ represents an action of type a with parameter 2. The only annotations in these examples are segmentations that cluster actions into groups that achieve a single non-primitive action, depicted by upper-case letters, such as X, Y, Z. The recipes contain both primitive and non-primitive action types, shown by lower and upper case letters, respectively. Dotted lines indicate optional steps. Arrows indicate ordering constraints. A Greek letter represents two or more parameters of actions that are bound to be equal by a recipe. A "$\cdot$" indicates that an action has a parameter that is not bound to any other parameter in that recipe.

represents partial ordering constraints on pairs of actions, all orderings of a, b, and c must be correct. Figure 4 shows an example of task model learning. The model learned after all three examples are processed accepts many action sequences that have not been seen, such as $[a(1), e(2), g(3), h(3), d(1,2)]$.

The ALIGNMENT function constructs a model m with non-primitive actions without parameters and recipes with only required steps. It also constructs an *alignment* from the annotated examples $\bar{\mathcal{E}}$ to m that consists of a pair of mappings $\langle\sigma, \phi\rangle$ where ϕ maps from each segment in each $e \in \bar{\mathcal{E}}$ to a recipe in m, and σ maps from each element in each segment in each $e \in \bar{\mathcal{E}}$ to a step in a recipe in m. These mappings are used by the induction algorithms.

ALIGNMENT first partitions the segments in $\bar{\mathcal{E}}$ into sets of segments that must, under our assumptions, be mapped to the same recipe. In particular, it groups the segments such that any two segments, s_i and s_j, are grouped together if they have the same *segmentType* and the set of the types of the elements in s_i that are not marked optional are a subset of the set of the types of the elements in s_j that are not marked optional. For any set of annotated examples, there is only one possible such grouping, which can be easily computed.

Next, for each group of segments, ALIGNMENT creates a recipe with n_t steps of each action type t, where n_t is the largest number of elements of type t that occur in any segment in the group. It maps each segment in the group to this recipe, and maps the segment's elements, in order of occurrence, to steps of the same type in the recipe.

After alignment, our algorithm determines the optionality of, and ordering constraints between, steps. The INDUCEOPTIONAL function marks step s in recipe r as optional if any segment element that is marked optional is mapped to s or if some segment is mapped to r but contains no element that is mapped to s. The INDUCEORDERING function adds a constraint that orders step s_i before step s_j unless there is a segment that contains elements e_i and e_j such that e_i is mapped to s_i and e_j is mapped to s_j and either e_j occurs before e_i or the annotations indicate that this ordering was possible.

No pseudo-code is given for ALIGNMENT, INDUCEOPTIONAL, or INDUCEORDERING since they are not the focus of this paper.

Inducing propagators We now present methods for inferring bindings, constraints, and parameters of non-primitive actions, which we will refer to collectively as *propagators*.

The role of propagators is to enforce equality relationships among the parameter values of primitive actions. For example, in a task model for cooking spaghetti marinara, the cooked pasta must be the same pasta to which sauce is later added. In contrast, different knives can be used to cut, say, the tomatoes and the mushrooms. These equality relations cross the boundaries of many actions and recipes, i.e. they are not local to any particular recipe.

The first step of learning propagators is to decide which parameters values should be forced to be

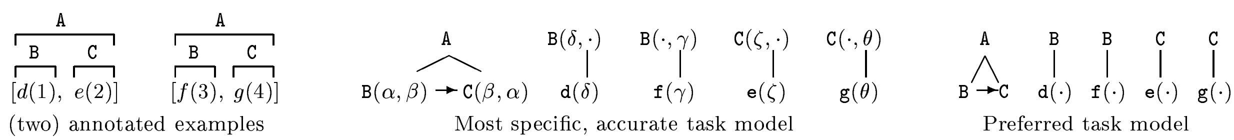

Figure 5: Motivation for suggested model bias. Both models can explain both examples. The task models differ in that only the first contains propagators that can force d and g's parameters and e and f's parameters to be equal. Without the suggested bias, the more elaborate model is preferred because the simpler model accepts examples (e.g., $[d(1), g(2)]$) which might not be valid examples from the domain.

equal. For example, if the same knife is used to cut vegetables in all examples, then we can conclude that the same knife must be used. If we then saw an example in which different knives were used, we would retract this constraint. Alternatively, the annotations can indicate that different knives could have been used.

A problem arises due to pairs of steps that never occur in the same example. As shown in Figure 5, there exist such task models that constrain the parameters of these steps to be equal. Such models are counter-intuitive because they postulate elaborate constraints that are not positively suggested by any example. Further, to remove all unnecessary constraints, the learning algorithm must see examples that contain all pairs of steps that are *unrelated* to each other.

To address this problem, we propose a bias against models with unsuggested propagators. Intuitively, an unsuggested propagator can be removed from a model and the model will still explain all seen examples. For example, in Figure 5, none of the propagators are suggested because no parameter values are equal in the two examples being modeled. However, in Figure 4, all of the propagators are suggested. We propose the following bias:

Suggested parameter preference bias: A model m_i is preferred to model m_j given annotated examples $\bar{\mathcal{E}}$ iff all of m_i's propagators are suggested by an example in $\bar{\mathcal{E}}$ and m_j has propagators that are not suggested by any example in $\bar{\mathcal{E}}$.

Model learning thus benefits from Occam's Razor: the simplest model that explains the data should be preferred. For propagators, we claim the simplest model contains only what is needed to explain the equalities evident in the examples.

Note that an unpreferred model may become preferred as more examples are seen. In Figure 5, if we saw [d(1),g(1)] and [f(1),e(1)] then all the propagators in the more elaborate task model would be suggested and so it would be preferred.

Figure 6 shows pseudo code for an algorithm for learning propagators with two modes, one that is sound with no preference bias and one that is sound under the suggested parameter bias. The algorithm takes as input the annotated examples and a task model that lacks some or all of these elements, and produces a more complete task model.

A data structure that is used to facilitate the computation of propagators is a *path*. A path starts at a parameter of a primitive action and "follows" a possibly empty sequence of recipe steps. Given a path p that has a non-empty sequence of steps, STEP(p) returns the last recipe step in the sequence, TAIL(p) returns a path identical to p except that STEP(p) is absent, and RECIPE(p) returns the recipe that contains STEP(p).

```
INDUCEPROPAGATORS (m,Ē) ≡
  forall R in ALLRECIPES(m)
    ADDCONSTRAINTS(R,Ē)
ADDCONSTRAINTS (R,Ē) ≡
  L ← ∅
  P ← NONRECURSIVEPATHSTORECIPE(R)
  forall p in P
    L ← L ∪ PATHPAIRINGS(p, P,Ē)
  forall L in L
    forall pairs p, p' in L
      name ← PROPAGATENAME(p,null)
      name' ← PROPAGATENAME(p',null)
      add a constraint between parameter name of STEP(p)
        and parameter name' of STEP(p')
PROPAGATENAME (p, inName) ≡
  tail ←TAIL(p)
  if tail has no steps
    then pName ← NAME(START(p))
    else
      pName ← GENSYM()
      PROPAGATENAME(tail, pName)
  if inName ≠ null
    R ← RECIPE(p)
    add a parameter named inName of type TYPE(START(p))
      to PURPOSE(R)
    add a binding between parameter inName of PURPOSE(R)
      and parameter pName of STEP(p) to R
  return pName
PATHPAIRINGS (p, P,Ē) ≡
  L ← ∅
  forall p' in P
    if PARAMETER(p) and PARAMETER(p') have
        never been negatively related in Ē
        and either p and p' have been positively related in Ē
          or the Suggested Parameter Bias is not in effect
      then L ← L ∪ {{p, p'}}
  return L
```

Figure 6: Pseudo code to infer propagators

The algorithm works by considering all pairs of paths that end at the same recipe $\mathcal{R}$. If the parameters at the start of these paths should always be constrained to be equal (the criteria for this depends on the preference bias), then a set of propagators are added to the task model to make sure this will be the case. The propagators are added in a top down fashion, first with a constraint on $\mathcal{R}$, and then recursively adding parameters to non-primitives and bindings to recipes that achieve them.

2.2 Empirical Results

Some experiments were run to better understand the tradeoff between how much information the expert provides in each example and how many examples must be provided. We have built an *oracle* to simulate a human expert that provides varying types of annotations. This approach focuses the results on this tradeoff rather than the best way to elicit annotations from the expert. At present, we do not presume that there is a data base of unannotated examples that either the expert or the learner can access — examples are generated by the expert as needed.

For our experiments, we start with a target task model and use it to simulate the activities of the domain expert, both in generating a sequence of primitive actions and in annotating them. After each example, we determine if the algorithm has produced a task model equivalent to the target task model given the examples it has seen. Additionally, we determine if each example was *useful*, i.e. if it contained any new information that was not implied the previous example, by seeing if the algorithm's internal data structures were altered.

We ran our experiments on two target task models. The first model represents part of a sophisticated tool for building graphical user interfaces, called Symbol Editor. The model was constructed in the process of developing a collaborative agent to assist novice users. The model contains 29 recipes, 67 recipe steps, 36 primitive acts, and 29 non-primitive acts. A typical example contains over 100 primitive actions. The second test model was an artificial cooking world model designed specifically to test the learning algorithm. The model contains 8 recipes, 19 recipe steps, 13 primitive acts, and 4 non-primitive acts. An example typically contains about 10 primitive actions. Both models have recursive recipes.

Segmentations and non-primitive action names are always provided by the oracle, but we varied whether the other annotations were provided. We ran all variations of possible combinations of annotation types, and report a subset in Table 1. In this table, the average and minimum are determined from the examples that produced a change to the underlying model; other examples are labelled "useless."

Annotation	Cooking			Symbol Editor		
	Avg.	Min.	Useless	Avg.	Min.	Useless
All	5.3	3	9.9	1.9	1	0.1
EOP	6.5	3	11.1	2.4	1	0.4
EP	7.2	4	14.1	3.0	2	0.5
EO	7.2	3	10.4	14.2	3	47.0
E	8.1	4	13.1	14.4	3	46.9
O	38.3	15	404.3	53.0	37	118.7
None	38.3	15	404.2	53.1	37	118.6

Table 1: The kind of annotations provided influences the number of examples needed to learn task models.

In Table 1, O indicates that all ordering annotations are given, E indicates that all equality annotations are given, and P indicates that all propagators are given. Annotating optional steps did not significantly impact the results except when all other annotations were given (indicated by 'All' in the table). The reason for this is that optionality is the easiest aspect to learn because it does not involve relationships between steps. Note that annotating equalities does not add any information when propagators are given. The data are the results of randomized sequences of examples — 100 trials for the cooking domain and 20 trials for the Symbol Editor.

The main surprise is that providing equality annotations dramatically reduces the number of required examples. This is interesting because it seems likely that it will be much less onerous for a human expert to indicate when apparent equalities in the example are coincidental, than to construct all the propagator information directly.

Another interesting result in Table 1 is that learning is strongly influenced by the order in which examples are processed. This is reflected both by the minimum number of required examples for any trial and the average number of useless examples per trial. One could imagine that a human expert would provide examples closer to the minimum than to the average and would not present as many useless examples.

3 Discussion

This section talks about the differences between the model produced by the learning techniques (that accurately reflects the domain) and a model that results in dialogs that a human user is comfortable with. At this stage of development, the impact of these differences can only be described anecdotally; a more rigorous evaluation would require human user studies.

The models that our techniques induce do not have semantically meaningful names for recipes or steps of recipes; it is an open problem to determine how to describe higher level actions, such as, for example, making a meal, from a series of lower level actions, such as cooking pasta and making sauce. We expect that a person will have to edit these learned models to give them names that describe activities in the domain for the user-agent collaboration. These names are then used by Collagen to allow the agent to participate in dialogues and collaborations with a user. However, even having meaningful names for actions is not sufficient to producing natural utterances on the part of an agent using the induced recipes. Language generation has long explored this matter, and Collagen uses a simple but effective set of generation mechanisms for changing recipe names and recipe steps into more natural text. As a result, a recipe name such as "Achieve MakeMeal" becomes "let's make a meal" when the agent proposes undertaking such an activity in a conversation with a user.

A similar problem arises for the names of parameters of non-primitive actions that are induced by the learning algorithms.[1] Our algorithm can make a reasonable guess for parameter names by using the type of the variable, such as assigning "kitchen" to the name of a parameter of type kitchen. However, there still might be several variables of the same type for a non-primitive action and automatically generated names like "kitchen1", "kitchen2", etc. will not be very meaningful.

A related issue are what can be called "semantic" variables. A semantic variable is a variable that is not needed in order to make the model accurate, but would improve the quality of the dialogs when an agent is acting from the model. For example, in our sample domain, the PreparePasta action does not need a parameter to indicate which pasta is being prepared. However, it is useful for PreparePasta to contain such a parameter so that the agent can refer to the pasta to be prepared, e.g., "Let's prepare that pasta on the counter".

Figure 7 shows a sample history when the agent is acting from an unedited learned task model. The model allows the agent to be an active participant only if the user is savvy enough to be able to interpret the agent's utterances. All of the deficiencies outlined above are present somewhere in Figure 7.

An area of future work is to apply the learning techniques to support the "online" refinement of task models during dialogue. For example, suppose the agent says "Let's make the sauce" and the user responds by taking out a can of clams. If the agent does not have a recipe for making clam sauce, it might say "Are you making sauce?". If the user says that she is, then the agent can observe the user's actions until the user moves onto the next subtask, at which point the agent can invoke the learning techniques described above to produce a new recipe. The agent might also ask a few questions to get more information about which steps were optional or what are the ordering constraints among steps. The same approach could be used to generalize an existing recipe: if the user does something unexpected after, say, the third step of a known recipe, the agent can ask if the user's action was part of achieving the current goal or not, and update the task model accordingly. The resulting model might contain a recipe with a new step or with fewer constraints. Finally, the task model can be improved if the user were to make statements such as "we didn't need to do that" or "we didn't have to use the same pan as we did before".

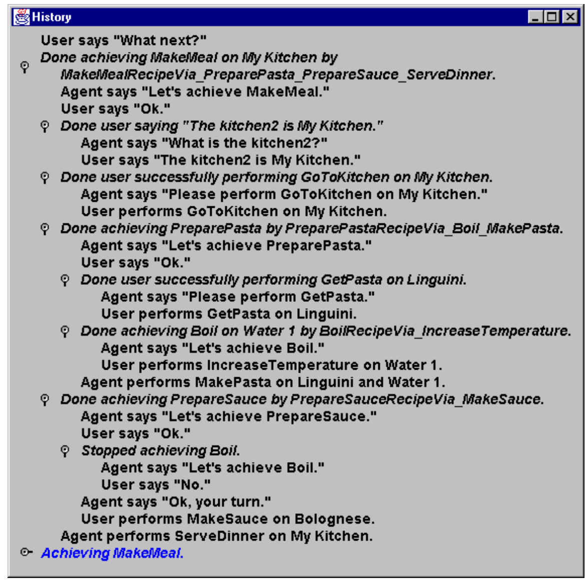

Figure 7: Sample history from learned model.

4 Related research

Much on the recent work on learning and adaptation in dialogue has focused on learning techniques for modeling aspects of dialogue and discourse. Walker (2000) and Litman et al. (2000) used reinforcement learning to optimize the dialogue strategy for conversational initiative on the part of a spoken language system while Chu-Carroll et al. (1997) used Dempster-Shafter Theory to predict initiative in dialogue. DiEugenio et al. (1997) used C4.5 learning to predict cue usage. Like our work, all of these efforts involve off line exploration of discourse behavior to train a language system to produce certain behaviors; however, our

[1] While the learning algorithm presented produces an inordinate number of propagators, the actual implementation reuses propagators when possible.

work is focused on the tasks that underlie discourse rather discourse-specific phenomena. The only on-line effort we are aware of is Göker and Thompson's (Göker and Thompson, 2000) adaptive agent, which proposes to use learning to develop a user model to track a user's preferences and thereby in real-time to avoid asking for information incompatible with those preferences.

Bauer (1998, 1999) presents techniques for acquiring non-hierarchical task models from unannotated examples for the purpose of plan recognition (i.e., inferring a person's intentions from her actions). Since the task model is used primarily for recognition, Bauer's algorithm learns only the required steps to accomplish each top-level goal. Bauer introduces heuristics for solving what we refer to as the alignment problem. (In contrast, we side-step the problem by restricting the task model language). Since our task models are intended to support collaboration and discussion of tasks, we found it important to extend Bauer's work to handle hierarchical task models and optional steps.

Tecuci et. al. (1999) present techniques for producing hierarchical if-then task reduction rules by demonstration and discussion from a human expert. The rules are intended to be used by a knowledge-based agent that assists people in generating plans. In their system, the expert provides a problem-solving episode from which the system infers an initial task reduction rule, which is then refined through an iterative process in which the human expert critiques attempts by the system to solve problems using this rule. Tecuci et. al. have not specifically addressed the problem of inferring parameters and bindings for intermediate goals.

Other research efforts have addressed aspects of the task model learning problem not addressed in this paper. Angros Jr. (2000) presents techniques that learn recipes that contain causal links, to be used for the intelligent tutoring systems, through both demonstration and automated experimentation in a simulated environment. Masui and Nakayama (1994) investigates learning macros from observation of or interaction with a computer user in order to assist the user with tasks that occur frequently or are inherently repetitive. Lau et. al. (2000), in one of the few formal approaches to learning macros, uses a version space algebra to learn repetitive tasks in a text-editing domain. Gil and Melz (1996) and Kim and Gil (2000) have focused on developing tools and scripts to assist people in editing and elaborating task models, including techniques for detecting redundancies and inconsistencies in the knowledge base, and making suggestions to users about what knowledge to add next.

References

Richard Angros Jr. 2000. Agents that learn what to instruct: Increasing the utility of demonstrations by actively trying to understand them. Technical report, Univ. of Southern California.

Mathias Bauer. 1998. Acquisition of Abstract Plan Descriptions for Plan Recognition. In *AAAI98*, pages 936–941.

Mathias Bauer. 1999. From Interaction Data to Plan Libraries: A Clustering Approach. In *Proc. of the 16th Intl. Joint Conf. on AI.*

J. Chu-Carroll and M. Brown. 1997. Tracking initiative in collaborative dialogue interactions. In *Proc. of the 35th Annual Conf. of the Association of Computational Linguistics*, pages 262–270.

Allen Cypher, editor. 1994. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA.

B. Di Eugenio, J.D. Moore, and M. Paolucci. 1997. Learning features that predict cue usage. In *Proc. of the 35th Annual Conf. of the Association of Computational Linguistics*, pages 80–87.

Y. Gil and E. Melz. 1996. Explicit representations of problem-solving strategies to support knowledge aquisition. In *Proc. 13th Nat. Conf. AI.*

Mehmet G. Göker and Cynthia Thompson. 2000. Personalized, conversational case-based recommendation. In *Advances in Case-Based Reasoning, Proceedings, 5th European Workshop on Case-Based Reasoning*, Trento, Italy.

Barbara Grosz and Candace Sidner. 1990. Plans for discourse. In Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, editors, *Intentions in Communication*, pages 417–444. MIT Press, Cambridge, MA.

J. Kim and Y. Gil. 2000. Aquiring problem-solving knowledge from end users: Putting interdependency models to the test. In *Proc. 17th Nat. Conf. AI*, pages 223–229.

Tessa Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version space algebra and its application to programming by demonstration. In *Intn. Conf. on Machine Learning*, pages 527–534.

Neal Lesh, Charles Rich, and Candy Sidner. 1999. Using plan recognition in human-computer collaboration. In *Proc. of the 7th Intl. Conf. on User Modeling*, pages 23–32.

N. Lesh, C. Rich, and C. Sidner. 2001. Collaborating with focused and unfocused users under imperfect communication. In *Proc. 9th Int. Conf. on User Modelling.*

D. Litman, S. Singh, M. Kerns, and M. Walker. 2000. NJFun: A reinforcement learning spoken dialogue system. In *Workshop on Conversational Systems, ANLP/NAACL Conference, 2000*, pages 17–20.

K. E. Lochbaum. 1998. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4), December.

T. Masui and K. Nakayama. 1994. Repeat and predict– two keys to efficient text editing. In *Conference on Human Factors in Computing Systems*, pages 118–123.

C. Rich, C. Sidner, and N. Lesh. 2001. Collagen: Applying collaborative discourse theory to human-computer interaction. *Artificial Intelligence.* Special Issue on Intelligent User Interfaces. To appear.

G. Tecuci, M. Boicu, K. Wright, S. W. Lee, D. Marcu, and M Bowman. 1999. An integrated shell and methodology for rapid development of knowledge-based agents. In *Proc. 16th Nat. Conf. AI.*

Marilyn A. Walker. 2000. An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email. *Journal of Artificial Intelligence Research*, 12:387–416.