

Learning Hierarchical Task Models by Defining and Refining Examples

Andrew Garland, Kathy Ryall, and Charles Rich

TR2001-26 December 2001

Abstract

Task models are used in many areas of computer science including planning, intelligent tutoring, plan recognition, interface design, and decision theory. However, developing task models is a significant practical challenge. We present a task model development environment centered around a machine learning engine that infers task models from examples. A novel aspect of the environment is support for a domain expert to refine past examples as he or she develops a clearer understanding of how to model the domain. Collectively, these examples constitute a test suite that the development environment manages in order to verify that changes to the evolving task model do not have unintended consequences.

Knowledge Capture, October 2001

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Submitted May 2001. Revised August 2001.

Learning Hierarchical Task Models by Defining and Refining Examples

Andrew Garland and Kathy Ryall and Charles Rich
Mitsubishi Electric Research Laboratories
{garland,ryall,rich}@merl.com

Abstract

Task models are used in many areas of computer science including planning, intelligent tutoring, plan recognition, interface design, and decision theory. However, developing task models is a significant practical challenge. We present a task model development environment centered around a machine learning engine that infers task models from examples. A novel aspect of the environment is support for a domain expert to refine past examples as he or she develops a clearer understanding of how to model the domain. Collectively, these examples constitute a “test suite” that the development environment manages in order to verify that changes to the evolving task model do not have unintended consequences.

Keywords

programming by demonstration, knowledge acquisition

INTRODUCTION

Many fields of computer science — planning, intelligent tutoring, plan recognition, interface design, and decision theory to name a few — get a lot of leverage from applying general-purpose algorithms to domain-specific task models. This approach gives rise to the notorious *knowledge acquisition bottleneck*: developing an accurate domain model is a significant engineering obstacle. In this paper, we present a development environment that can ease the task model acquisition process. The environment combines direct model editing, machine learning based upon annotated examples, and model verification through regression testing. The learning techniques and most of the other major components of this environment are in place; however, the graphical front-end is still under development.

In this work, the problem of developing task models is considered in the context of the Collagen [18, 17] system. In Collagen, a collaborative interface agent engages in dialogs with a user to jointly achieve tasks. Collagen is an implementation of the SharedPlan theory of collaborative discourse [7], in which the agent’s behavior is driven by general-purpose

algorithms for discourse interpretation [13], plan recognition [10], and action selection [11]. In order to apply these algorithms in a given domain requires constructing an explicit, declarative model of the underlying task structure.

Since Collagen task models are hierarchical, a domain expert must decide how to divide tasks into subtasks, which involves choosing the best abstractions to represent 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. Determining an appropriate set of intermediate goals (as well as the number and type of parameters for each) can be extremely difficult for a domain expert.

Our approach to acquiring task models is based on the conjecture that it is often easier for people to generate and discuss *examples* of how to accomplish tasks than it is to deal directly with task model abstractions. In a sense, we designed a kind of programming by demonstration [4, 12] system in which a domain expert performs a task by executing actions and then reviews and annotates a log of the actions.

In prior research, we developed machine learning techniques that infer hierarchical task models from a set of partially-annotated examples of task-solving behavior [5]. As a general tradeoff, an expert can provide minimal annotations about many examples or more exhaustive annotations about fewer examples. Also, as will be discussed below, certain types of annotations are more valuable to the learning engine.

We have integrated these machine learning techniques into a development environment that provides comprehensive support for experts to generate task models. This involves removing or refining past examples as well as defining new examples. In addition to learning from the collection of examples, the system can use them for regression testing to verify the behavior of the model throughout the development process. This technique detects more potential errors than simply checking the internal consistency of a model.

The design presented in this paper reflects the collective experience of the Collagen research group over the past several years “manually” developing task models. Typically, a model is constructed through an incremental development process, which is described in the following two paragraphs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

K-CAP’01, October 22-23, 2001, Victoria, British Columbia, Canada.

Copyright 2001 ACM 1-58113-380-4/01/0010... \$5.00

Initial versions of a task model are inferred from a small number of examples that show the most common solutions to key domain tasks. Both the model and the examples frequently undergo substantial revisions during this early stage. Next, the model will be generalized to cover additional examples that demonstrate solutions involving, for example, alternate orderings for actions, optional behavior, or alternate task decompositions. Occasionally, defining additional examples will spur the expert to re-conceptualize the entire domain, necessitating reworking many previous examples.

As the development process nears completion, there is less and less benefit to providing new examples. It is generally faster and easier to directly edit the model. Also, learned models, even when accurate, may need to be tweaked by the expert for other reasons. For example, as discussed in [5], the organization of a complete and accurate task model may be inappropriate for a collaborative agent. It is at this final stage of development that the ability to easily verify the behavior of the model using the collection of past examples is critical.

The next section of the paper describes how task models are inferred from the annotated examples of the expert; we also provide empirical results based on our implementation of this learning module. The third section of the paper presents the design of the model building environment in detail. The paper concludes with a discussion of related research.

MACHINE LEARNING FROM EXAMPLES

Within the task model development environment, there is a division of labor between the user and the computer: the user provides annotated examples so that the learning system can generalize the task model under development.

This section describes how a domain expert partially annotates examples. We describe the task model language first and then the different types of partial annotations. 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 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. Actions do not currently include an explicit representation for preconditions and effects.

Recipes are methods for decomposing non-primitive actions. Each recipe specifies a set of steps that are performed to achieve the non-primitive action that is the collective objective of the steps. All steps are assumed to be required unless they are labelled as optional. There may be several different recipes for achieving a single non-primitive action.

A recipe also contains constraints that impose partial temporal orderings on its steps, as well as various logical relations

among their parameters. For the purposes of this paper, the only logical relations we will consider are equalities. Equalities between a parameter of a step and a parameter of the objective of the recipe are called bindings, but are otherwise indistinguishable from constraints. Parameters and steps have a name as well as a type in order to allow for unambiguous references (in bindings and constraints) to multiple steps of the same type.

Figure 1 contains samples of this representation for a cooking domain that will be used throughout this paper as a running example. This domain was chosen over alternate Collagen task models because it is intuitive and can be easily varied in order to conduct empirical studies. A task model in the form of Figure 1 is the desired output of learning.

```

nonprimitive act PreparePasta
  parameter Pasta pasta

primitive act GetPasta
  parameter Pasta pasta

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

```

Figure 1: Collagen representations from a cooking domain (keywords are in bold).

Annotation Language

Informally, the input to the learning algorithm is a series of demonstrations; each one explicitly shows one correct way to perform a task and, via annotations, indicates other similar ways that are also correct. 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 represents partial ordering constraints on pairs of actions, all orderings of a, b , and c must be correct.

More precisely, each input to the learning engine is an annotated example e , where e is a five-tuple: $(\hat{e}, \mathcal{S}, \text{optional}, \text{unordered}, \text{unequal})$:

\hat{e} is the temporally ordered list of actions $[p_1, \dots, p_k]$ that constitute the *unannotated* example demonstrated by the expert. In most cases, each p_i will be a primitive action; however, p_i could also be an intermediate goal. The semantics of the latter case is that p_i is being used as a placeholder in lieu of fleshing out the example to include a segment that achieves p_i .

\mathcal{S} is a segment, which is a pair $\langle \text{segmentType}, [s_1, \dots, s_n] \rangle$. Each s_i , called a *segment element* or element for short, is either an action or a segment. Grouping elements together means that they collectively achieve a non-primitive act of type *segmentType*.

optional is a partial mapping from elements to boolean values. If the mapping is defined and is true, the expert is specifying that removing that segment 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. This mapping does not convey information about inequality relations; i.e. this mapping cannot indicate that two parameters must never have the same value.

Figure 2 contains an example of how this formal notation is used to define an annotated example in the cooking domain. Each 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.

```

 $\hat{e} = [ \text{Boil}_1(\text{water}_9), \text{GetPasta}_2(\text{spaghetti}_4),
      \text{CookPasta}_3(\text{spaghetti}_4, \text{water}_9), \text{Boil}_4(\text{water}_9),
      \text{MakeSauce}_5(\text{marinara}_7), \text{ServeDinner}_6(\text{kitchen}_5) ]$ 

 $\mathcal{S} = \langle \text{MakeMeal}_9, [
  \langle \text{PreparePasta}_7, [\text{Boil}_1, \text{GetPasta}_2, \text{CookPasta}_3] \rangle
  \langle \text{PrepareSauce}_8, [\text{Boil}_4, \text{MakeSauce}_5] \rangle
  \text{ServeDinner}_6 \rangle$ 

 $\text{optional}(\text{GetPasta}_2) = \text{true}$ 
 $\text{optional}(\text{ServeDinner}_6) = \text{false}$ 
 $\text{unordered}(\text{PreparePasta}_7, \text{PrepareSauce}_8) = \text{true}$ 
 $\text{unordered}(\text{Boil}_1, \text{CookPasta}_3) = \text{false}$ 
 $\text{unequal}(\text{Boil}_1.\text{water}, \text{Boil}_4.\text{water}) = \text{true}$ 
 $\text{unequal}(\text{GetPasta}_2.\text{pasta}, \text{CookPasta}_3.\text{pasta}) = \text{false}$ 

```

Figure 2: Sample annotated example, in formal notation.

Figure 2 defines one top-level segment of type *MakeMeal*, which is composed of two sub-segments (*PreparePasta*, *PrepareSauce*) and a primitive action (*ServeDinner*). The partial mappings at the bottom of the figure indicate that *GetPasta* is optional and *ServeDinner* is required. Also, the steps of type *PreparePasta* and *PrepareSauce* may appear in any order in general, while the *Boil* step of *PreparePasta* must always precede the *CookPasta* step. Finally, the annotations indicate that the com-

mon water parameter value for *Boil*₁ and *Boil*₄ is a coincidence, but that the pasta parameter of *GetPasta* and *CookPasta* will always be the same.

A graphical interface, which is part of the development environment, allows experts to annotate this information in a more intuitive way, such as by marking certain nodes in a tree visualization. However, it should be clear from this partially annotated example that fully annotating examples would be quite burdensome regardless of the interface.

It is tempting to draw conclusions from the absence of certain annotations. For example, in Figure 2, many steps are not marked as optional. While one might interpret 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. To handle such cases, the learning techniques distinguish between positive evidence (e.g., annotating that a step is required) and the lack of negative evidence (e.g., the step appears in all defined examples involving this recipe).

A key feature of our learning algorithm is that it infers 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 the marinara 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.

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 to learn an accurate model of the domain. For testing purposes only, we 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.

In each experiment, we start with a target task model and use it to simulate the activities of a domain expert, both to generate unannotated examples and to annotate them. Segmentations and non-primitive action names are always provided by the simulated expert, but we varied which other annotations were provided. After each example is input to the learning engine, we determine if the generalized task model is equivalent to the target model. Also, we determine if each example was “useful,” i.e. if it contained any information that altered the contents of the data structures used for inference; other examples are labeled “useless.”

We ran experiments on two target task models. The first represents part of a sophisticated tool for building graphical user interfaces, called the Symbol Editor. The model was constructed in the process of developing an agent to assist novice users of the Symbol Editor. 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 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.

We ran all variations of possible combinations of annotation types, and report a subset in Table 1. In this table, O indicates that all ordering annotations are given, E indicates that all equality annotations are given, and P indicates that all propagators are given (propagator annotations subsume equality annotations). Optional steps are only annotated when all annotations are 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. The data are the results of randomized sequences of examples — 100 trials for the cooking domain and 20 trials for the Symbol Editor. Also, the average and minimum are measured on useful examples.

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

Annotation	Cooking			Symbol Editor		
	Avg.	Min.	Useless	Avg.	Min.	Useless
All	5.3	3	9.9	1.9	1	0.1
OP	6.5	3	11.1	2.4	1	0.4
P	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

The main surprise is that providing equality annotations dramatically reduces the number of required examples (from 38.3 to 8.1 for cooking). This is encouraging 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 (which is roughly half the average number) of useful examples and the average number of useless examples (which is comparatively large). It is possible a human would provide diverse, useful examples so that the number of examples required in practice would be close to the minimum number of useful examples.

DESIGN FOR A MODEL DEVELOPMENT ENVIRONMENT

This section presents the design for our task model development environment. A novel aspect of the environment is support for a domain expert to refine past examples as he or she develops a clearer understanding of how to model the domain. Collectively, these examples constitute a “test suite” that the development environment manages in order to verify that changes to the evolving task model do not have unintended consequences.

Figure 3 shows an idealized sequence that a user would follow to develop a task model. In practice, a model would not be developed in such a straightforward path. For example, the model can be inferred, visualized, or manually edited by the user at arbitrary points during the development cycle.

-
- Define a starting set of actions (optional).
 - Define examples.
 - Generalize the model based on the examples.
 - Visualize the resulting model.
 - Use the model in subjective tests of quality.
 - Refine prior examples.
 - Manually edit the task model.
 - Run regression tests with the collection of examples.

Figure 3: Typical steps in task model development

Most of the process described in Figure 3 is presently achievable through a single GUI application; however, certain aspects currently require a combination of command-line programs and text-file editing. Fully integrating the current capabilities into a single tool will reduce the burden for the domain expert, and will provide opportunities for providing more assistance.

Both Collagen and the model building tool are written in the Java programming language. This has two important beneficial consequences. First, it is easy to design the system to switch between alternate components (e.g., model viewers or learning engines) by using interfaces and Java Beans. Second, the task model language for Collagen is implemented as a superset of Java, which permits very specific refinements of task models for any particular domain.

The rest of this section is a “story board” that illustrates how a person might use our system to develop a task model for making a meal. As this task is something that occurs in the physical world, the user constructs examples by virtually walking through the process of making a meal — it is a mental walk-through, rather than an actual walk-through. In contrast, for a computer application with a graphical user interface that has already been built and implemented, a person could simply run the application, and annotate the resulting log. For an application that is being (re-)designed, an expert would use the virtual walk-through process to create examples.

Define a Starting Set of Actions (Optional) The first step the user may take is to generate an initial list of primitive and non-primitive acts, as shown in Figure 4.

Non-Primitives: MakeMeal
 Primitives: Boil, CookPasta, PrepareSauce, ServeDinner

Figure 4: Initial working set of action types

This categorization may change over time, but helps to bootstrap the process. Our system does not assume the existence of a pre-defined hierarchy of actions (i.e., an ontology) for the domain — determining this hierarchy is a major part of task model development. While defining the primitives for an implemented GUI application may be straightforward, for activities in the real world the process is more difficult. In all non-trivial domains, identifying the correct set of abstract (non-primitive) actions is challenging.

Define Examples An example is an annotated list of instantiated actions (action type plus specific values for parameters) that constitute the achievement of a goal in the domain. The user may start by constructing an unannotated demonstration of how to make a meal, as shown in Figure 5 (for reading ease, actions in the examples of this section are not subscripted).

GetPasta
 Boil
 CookPasta
 PrepareSauce
 ServeDinner

Figure 5: First example, unannotated

In Figure 5, the user has not specified any parameter values. Also, the example includes an unknown action type (GetPasta), so the system may either ask the user if the action should be added to the working set as a primitive act or silently do so, depending on a settable option.

Working with this example, the user groups related actions into segments; for each segment, the user provides a name that describe the purpose of the segment. In the minimally annotated version shown in Figure 6, the elements of a segment are identified visually by the level of indentation — the purpose name for a segment, which precedes its elements, is surrounded by brackets. In this case, the user has grouped the first three steps into `PreparePasta`. The system recognizes that this act is not part of the working set and can ask if it should be added to the non-primitives. In practice, annotations don't have to be added during a second pass — they can be done at the same time that actions are added.

Generalize the Model Based on the Examples After annotating one or more examples, the user can invoke the infer-

```
[MakeMeal]
  [PreparePasta]
    GetPasta
    Boil
    CookPasta
  PrepareSauce
  ServeDinner
```

Figure 6: First example, minimally annotated

ence engine to generalize the task model to incorporate the examples. Even if the learning tool has only one example to process (i.e. this example is the first one submitted by the user), generalization may occur if the same non-primitive action-type appears more than once in the example.

The result of learning from the example in Figure 6 is given in Figure 7. A comparison with Figure 1 shows that the propagators are missing from this first version of the task model. Barring guidance from the domain expert, the automated techniques name each recipe based on the types of the required steps, sorted alphabetically, of the recipe. Also, step names are derived from the type of the step.

```
nonprimitive act PreparePasta
primitive act GetPasta
recipe Boil_CookPasta_GetPasta achieves PreparePasta
  steps      Boil boil
              CookPasta cookPasta
              GetPasta getPasta
constraints  getPasta precedes boil
              boil precedes cookPasta
```

Figure 7: A portion of the task model after one example

An important issue is how the machine learning techniques will preserve manual edits to the task model. This issue arises in many environments where a human and a computer are collaborating — a person will often want to pin down some parts of the problem at hand so that the computer does not modify them when formulation a solution. The learning techniques currently allow an expert to pin down some parts of the system, but there are still some open issues regarding parameters for non-primitives.

As discussed in the previous section, the learning techniques free the domain expert from having to specify non-primitive parameters. Or, for those who are so inclined, an expert can specify the number and type of parameters for each non-primitive. However, mixing the two approaches requires either heuristics or dialogs with the expert or a combination of both. To see why, imagine an expert state that a non-primitive has a parameter of type X and that the learning engine infers the need for two slots of type X. Do either of these get mapped to the user-specified parameter and, if so, which one? There

are other ambiguous situations where the decision whether or not to re-use a propagator is unclear.

Define Additional Examples The expert iterates through this process until the task model is complete enough to test subjectively. After the user defines each additional example, inconsistencies between the current model and the new example need to be worked out. There are several ways in which the user and the system could interact to resolve inconsistencies; currently it is the sole responsibility of the domain expert.

As a result of resolving inconsistencies, sometimes the model will be changed and sometimes the example will be changed. For example, an action that was originally defined as a primitive action might appear as a segment purpose type. Sometimes the definition of the action needs to be changed and sometimes the expert makes an error. Another common source of inconsistencies is in the number and type of parameters for an action.

Figure 8 is an example of making linguini with clam sauce that might be provided as a second example to the learning system. In this figure, the user has added more detail by decomposing `PrepareSauce` and specifying parameters.

```
[MakeMeal]
  GoToKitchen(kitchen2)
  [PrepareSauce]
    Boil(water3)
    CookClams(clams8, water3)
    MakeClamSauce(clams8)
  [PreparePasta]
    Boil(water3)
    CookPasta(linguini11, water3)
  ServeDinner(kitchen2)
```

Figure 8: Second example, minimally annotated

When processing this second example, the system adds new acts to the working set of primitives, moves `PrepareSauce` from the primitives to the non-primitives, and adds parameters to the definitions of actions that appear in this example. The system also determines that there are optional steps in two different recipes (going to the kitchen and getting the pasta) and that the `PreparePasta` and `PrepareSauce` steps are unordered. Finally, in this example, the water used in preparing the sauce and the pasta happen to be the same so the system infers a set propagators that will force this equality to always hold. Figure 9 shows the output of learning.

In future examples, new acts may be demonstrated. Or, additional recipes to achieve known acts may be shown (e.g., achieving `MakeMeal` by calling a take-out restaurant). Also, additional orderings will be learned for known recipes. Other examples will show that parameters are not tied to specific domain literals; likewise, the water used in preparing a sauce and the water used in preparing pasta are generally differ-

```
nonprimitive act PreparePasta
  parameter Water water

primitive act GetPasta

recipe Boil_CookPasta achieves PreparePasta
  steps
    Boil boil
    CookPasta cookPasta
  optional GetPasta getPasta
  bindings
    achieves.water = cookPasta.water
  constraints
    boil.water = cookPasta.water
    cookPasta.water = water3
    cookPasta.pasta = linguini11
    getPasta precedes boil
    boil precedes cookPasta
```

Figure 9: A portion of the task model after two examples

ent. Future work includes investigating mechanisms for the learning system to indicate what types of examples would most benefit the learning process.

Visualize the Model After the inference techniques have generalized the model to account for this example, the expert can review the result to see if it matches his or her intention. In future work, additional bookkeeping by the learning techniques will enable the expert to find out what part(s) of which example(s) implied various pieces of the model (e.g. step optionality, ordering and equality constraints). Some pieces of the model will not be tied to examples — for example, if the model is manually edited.

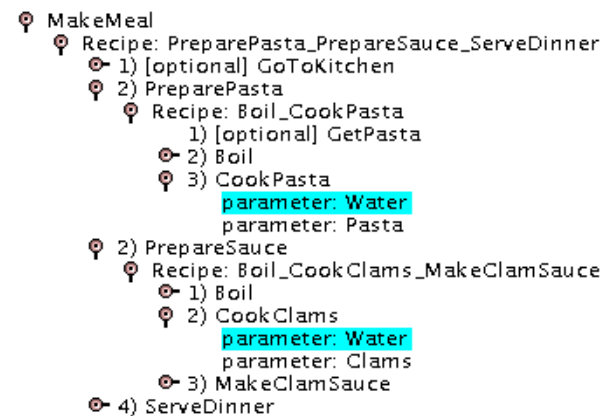


Figure 10: A graphical view of part of a task model.

Another feature of our system is that it supports multiple views of the task model. In Figure 1 (and throughout this section) we saw the textual representation of a portion of a task model. In Figure 10 we see a graphical view of the decompositions for a task in the task model, drawn as a tree.¹ The children of non-primitive actions are the recipes that achieve that act; a recipe expands to show the steps of that recipe; and

¹To simplify exposition, some visual elements of the tool have been suppressed or replaced by text (e.g. “[optional]”).

the children of primitive acts are the action type's parameters. The numbering of recipe steps summarizes the precedence relations. Also, parameters that are constrained by the model to always be equal, i.e. the two `Water` parameters shown, are indicated by having the same background color (different colors are used for each set of propagators).

Use the Model in Subjective Tests of Quality Ideally, the current task model can be evaluated by the domain expert by interacting with a collaborative agent for an existing GUI application.² Based on this interaction, the expert can identify weaknesses or errors in the model. For example, the expert can notice when the agent erroneously propagates a parameter value (as would be the case for the water used to boil clams and pasta). To improve the quality or accuracy of the model, the expert can create new examples, refine old examples, or manually edit the model .

Refine Prior Examples After using the task model (or at other times), the domain expert may wish to refine prior examples. For example, showing the learning system that `GetPasta` has a pasta parameter with the same value as `CookPasta` is easily done by adding parameter values to the first example. Figure 11 shows such a refinement of the first example.

```
[MakeMeal]
  [PreparePasta]
    GetPasta(ziti4)
    Boil(water12)
    CookPasta(ziti4, water12)
  [PrepareSauce]
  ServeDinner(kitchen1)
```

Figure 11: First example, refined with parameters

In Figure 11, `PrepareSauce` is now marked as a placeholder non-primitive, i.e. a non-primitive that is not decomposed in this example. Alternately, the domain expert could have refined `PrepareSauce` in Figure 11.

Manually Edit the Task Model The domain expert may wish to manually edit the task model, perhaps as a result of a subjective evaluation. For example, using the task model inferred from the two defined examples reveals that some dialogs flow unnaturally because `PreparePasta` does not have a parameter of type `Pasta`, even though that parameter is not needed for correctness. In addition, some experts may choose to replace the automatically generated recipe and step names, or may choose to take advantage of Collagen's flexible glossing (English text generation) mechanism.

Since the task model language for Collagen is a superset of Java, actions and recipes can include arbitrary Java code. In terms of system design, this means that there must be support

²Evaluation is always possible — Collagen can simulate the behavior of a collaborative agent in the absence of an existing application.

for experts to manually add Java code to the task model when needed.

Regression Testing As task models become complex, it is easy to make changes that have unexpected consequences. Our current implementation incorporates a rigorous testing facility to identify when changes to the task model influences the analysis of stored examples. Regression testing is not as useful in the early stages of model development, when action and recipe definitions are undergoing rapid change.

As the model is developed and refined, older examples may become obsolete. Changes in the number and types of parameters for an action, for example, may cause an early example to no longer be consistent with the current version of the task model. Likewise, as actions are added and deleted, and potentially reclassified as primitives or non-primitives, earlier examples may no longer be valid. An important consideration for the development environment is how to alert the user that these examples exist and how to provide an explanation of why the example is no longer usable. In some cases, the user may choose to disregard an early example while at other times he or she may choose to update an earlier example so that it can continue to be used in regression testing.

RELATED RESEARCH

Tecuci *et al.* [19] present techniques for acquiring large numbers of hierarchical if-then task reduction rules through demonstration by and discussion with a human expert. 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.* do not specifically address either the issue of building a model in the absence of a pre-defined ontology or the notion of regression testing to ensure that model updates preserve correctness.

Gil and Melz [6] and Kim and Gil [8] have reported on a wide range of issues related to building knowledge acquisition tools for developing databases of problem-solving knowledge. In contrast to our approach of inferring task models from annotated examples, they 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, as well as making suggestions to users about what knowledge to add next.

Paternò [15, 16] presents a graphical tool for eliciting hierarchical task models, represented as `ConcurTaskTrees`, of cooperative activities. This tool provides support for converting informal scenarios into formal descriptions and then verifying the consistency of a `ConcurTaskTree` with saved scenarios. In addition to some key differences in representation language (such as alternate decompositions for abstract actions and the ability to enforce arbitrary constraints among parameters), this tool does not support inference.

Other research efforts have addressed aspects of the task model learning problem not addressed in this paper. Bauer [2, 3] 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). OBSERVER [21] automatically learns the preconditions and effects of planning operators from unannotated expert solution traces and then refines the operators through practice. In a related approach, van Lent and Laird [20] present techniques to learn the preconditions and goal conditions for a hierarchy of operators (encoded as specialized Soar production rules) given expert-annotated performance traces.

Angros Jr. [1] 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 [14] investigate 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.* [9], in one of the few formal approaches to learning macros, uses a version space algebra to learn repetitive tasks in a text-editing domain.

CONCLUSION

This paper presented an approach, which is implemented in a development environment, for constructing and maintaining a hierarchical task model from a set of annotated examples provided by a domain expert. The key pieces of the system are a machine learning inference engine and a facility for conducting regression testing in order to verify the consistency of a task model with previously defined examples. As a general tradeoff, the domain expert can either provide minimal annotations about many examples or more exhaustive annotations about fewer examples.

Future work for this project fall into two broad categories: extensions to the inference engine and improvements in the usability of the development environment. One area for future work that falls into both categories is to develop *constructive critics*, i.e. algorithms that propose possible annotations, and include a facility for users to easily manage the advice provided by them. For example, one constructive critic might analyze past usage logs (or annotated examples) to suggest the segment elements that should be marked optional in an as-yet unannotated example. The user should be able to: 1) review such suggestions at any time, 2) easily understand them, and 3) easily accept none, some, or all of the them.

REFERENCES

1. R. Angros Jr. *Learning What to Instruct: Acquiring Knowledge from Demonstrations and Focussed Experimentation*. PhD thesis, University of Southern California, 2000.
2. M. Bauer. Acquisition of Abstract Plan Descriptions for Plan Recognition. In *Proc. 15th Nat. Conf. AI*, pages 936–941, 1998.
3. M. Bauer. From Interaction Data to Plan Libraries: A Clustering Approach. In *Proc. 16th Int. Joint Conf. on AI*, pages 962–967, 1999.
4. A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1994.
5. A. Garland, N. Lesh, and C. Sidner. Learning Task Models for Collaborative Discourse. In *Proc. of Workshop on Adaptation in Dialogue Systems, NAACL '01*, pages 25–32, 2001.
6. Y. Gil and E. Melz. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proc. 13th Nat. Conf. AI*, pages 469–476, 1996.
7. B. Grosz and C. Sidner. Plans for discourse. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 417–444. MIT Press, Cambridge, MA, 1990.
8. J. Kim and Y. Gil. Acquiring problem-solving knowledge from end users: Putting interdependency models to the test. In *Proc. 17th Nat. Conf. AI*, pages 223–229, 2000.
9. T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *Proc. 17th Int. Conf. on Machine Learning*, pages 527–534, 2000.
10. N. Lesh, C. Rich, and C. Sidner. Using Plan Recognition in Human-Computer Collaboration. In *Proc. of the 7th Int. Conf. on User Modeling*, pages 23–32, 1999.
11. N. Lesh, C. Rich, and C. Sidner. Collaborating with Focused and Unfocused Users under Imperfect Communication. In *Proc. 9th Int. Conf. on User Modeling*, pages 64–73, 2001.
12. H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.
13. K. E. Lochbaum. A Collaborative Planning Model of Intentional Structure. *Computational Linguistics*, 24(4):525–572, Dec. 1998.
14. T. Masui and K. Nakayama. Repeat and predict—two keys to efficient text editing. In *Conference on Human Factors in Computing Systems*, pages 118–123, 1994.
15. F. Paternò and C. Mancini. Developing task models from informal scenarios. In *Proc. ACM SIGCHI '99, Late-Breaking Results*, pages 228–229, 1999.
16. F. Paternò, G. Mori, and R. Galiberti. CTTE: An environment for analysis and development of task models of cooperative applications. In *Proc. ACM SIGCHI '01, Extended Abstracts*, pages 21–22, 2001.
17. C. Rich and C. Sidner. COLLAGEN: A Collaboration manager for Software Interface Agents. *User Modeling and User-Adapted Interaction*, 8(3/4):315–350, 1998.
18. C. Rich, C. Sidner, and N. Lesh. Collagen: Applying Collaborative Discourse Theory to Human-Computer Interaction. *AI magazine*, 22(4), 2001. To appear. <http://www.merl.com/papers/TR2000-38/>.
19. G. Tecuci, M. Boicu, K. Wright, S. W. Lee, D. Marcu, and M. Bowman. An integrated shell and methodology for rapid development of knowledge-based agents. In *Proc. 16th Nat. Conf. AI*, pages 250–257, 1999.
20. M. van Lent and J. Laird. Learning hierarchical performance knowledge by observation. In *Proc. 16th Int. Conf. on Machine Learning*, pages 229–238, 1999.
21. X. Wang. Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proc. 12th Int. Conf. on Machine Learning*, pages 549–557, 1995.