Interactively Defining Examples to be Generalized

Andrew Garland
Mitsubishi Electric Research Laboratories
garland@merl.com

Abstract

In a programming by demonstration system, examples provided by the user are generalized to form abstractions that are used by the system. There are several reasons why the system and the user should collaborate to annotate examples with knowledge that guides generalization. These include dealing with inconsistencies among examples or the underlying domain theory, making queries in order to speed generalization (e.g., active learning), and making suggestions (perhaps based on data mining) to change the outcome of generalization. This paper presents the design of a system that interacts with users via a set of "guessers": algorithms for suggesting possible annotations to the user, in the context of learning hierarchical task models.

Keywords

programming by demonstration, knowledge acquisition

INTRODUCTION

In programming by demonstration (PBD) systems [7, 14], examples provided by the user are generalized to form abstractions that are used by the system. This generalization process is guided by knowledge that indicates which parts of an example should be generalized, and how to generalize those parts. In this paper, we call the combination of the example provided by the user and the generalization knowledge about the example an *annotated* example.

In some systems [21, 11, 12], the user is responsible for generalizing the example. This simplifies the system, but can be very difficult on the user. At the other end of the spectrum are systems that use artificial intelligence techniques to make inference based on multiple examples [15, 6, 16, 9], possibly without user guidance. This benefits users since they need neither understand the generalization process nor annotate examples, but is not feasible in many practical situations.

The most practical approach lies in the middle ground, where the system and the user collaborate to annotate examples, since users and PBD systems have nearly disjoint, complementary competence areas [3]. This paper presents the design for a system that interacts with the user via a set of *guessers*. Guessers are algorithms that suggest possible annotations, which the user either "takes" to add the generalization knowledge or ignores.

In our framework, guessers derive suggestions from diverse knowledge sources. A guesser could encapsulate a single previous example, the current generalization of past examples, general pieces of domain theory, or a collection of (possible noisy and outdated) raw data. They propose interactions to resolve inconsistencies, to speed generalization, or to change the outcome of generalization. Most PBD inference techniques can be viewed as highly accurate, implicit guessers whose guesswork is out of the control (and possible out of sight) of the user. By contrast, an explicit guesser can suggest an annotation that is less likely to be accurate, since the user controls whether the annotation is made.

Interacting with guessers improves the quality of both the user experience and the generalization produced by the system. The user experience improves because the environment involves the user in the generalization process in a very flexible way. When involved, the user can better understand the problem (and the solution) on which the user and the system are jointly working. Yet, if a user feels overtaxed, he can reduce his involvement by ignoring many or all of the suggestions (suggestions do not demand the user's focus of attention). The quality of the generalization improves because the user better understands the annotation task and provides more annotations.

In our work, the guessers are organized into a committee, and the suggestions proposed to the user are those "passed" by the committee. This helps prevent information overload, where the user has to wade through many (possibly conflicting) suggestions. Second, as in *query by committee* [20], as long as we ensure a minimum competency of the members, the committee suggestions are more likely to be acceptable to the user than those of any individual guesser.

In this paper, interacting with a committee of guessers is studied in the context of learning a hierarchical task model, a data structure used in many fields of computer science. An existing task model development environment combines direct model editing, PBD based on partially annotated examples, and model verification through regression testing [9]. The current research integrates guessers into the example-

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

annotating environment and shows how this can simplify the task for the user.

The next section of the paper describes prior work to infer a hierarchical task models from annotated examples provided by the user. The third section of the paper describes how the user and the guessers interact when annotating examples. After a discussion and related research section, the paper concludes with a brief description of future work.

LEARNING HIERARCHICAL TASK MODELS

Previous work presented a software environment in which programming by demonstration was one piece in the overall process to develop hierarchical task models [9]. This section describes the representation and inference algorithms used by the PBD interface in which a user defines and partially annotates a set of examples. The next section shows how this task is simplified for the user when guessers are integrated into the interface.

Many fields of computer science — e.g., planning, intelligent tutoring, plan recognition, and interface design — gain leverage by applying general-purpose algorithms to domain-specific task models. However, developing an accurate domain model is an engineering obstacle dubbed the *knowledge acquisition bottleneck*. In sum, developing task models is an important and complex knowledge acquisition problem.

In this work, the knowledge representations and algorithms for inferring task models are motivated by the Collagen system [18, 17]. In Collagen, a collaborative interface agent engages in dialogs with a human to jointly achieve tasks. Collagen task models are hierarchical, where tasks are subdivided into subtasks, so that the agent can discuss how to accomplish tasks in a way that is intuitive to the human.

A Collagen task model is composed of actions and recipes. Actions are either primitive, which can be executed directly, or non-primitive (also called "intermediate goals"), 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 has an objective, which is a non-primitive act, and specifies a set of steps to perform to achieve its objective. 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 temporal orderings on its steps, as well as various logical relations among their parameters. For the purposes of this paper, the only constraints considered are equalities. Steps (parameters) have a name as well as a type in order to allow for unambiguous references to multiple steps (parameters) of the same type.

Figure 1 contains samples of this representation for a domain that will be used throughout this paper as a running example. This domain is for an existing application that is a gas turbine engine simulator used for virtual training [19]. A task model in the form of Figure 1 is the desired output of learning.

```
nonprimitive act OpenFuelValves
  parameter Engine gte

primitive act OpenFuelValve1
  parameter Engine gte

recipe OpenFVRecipe achieves OpenFuelValves
  steps        OpenFuelValve1 openfv1
               OpenFuelValve2 openfv2
  constraints  achieves.gte = openfv2.gte
               openfv1.gte = openfv2.gte
```

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

Partial annotations and learning

Within the task model development environment, there is a division of labor between the user and the computer: the user provides annotated examples and the learning system generalizes the task model. An expert can provide minimal annotations about many examples or more exhaustive annotations about fewer examples. Some annotations provide more useful information to the learning engine than others.

This subsection describes how a domain expert partially annotates examples and how the learning engine infers a task model from them. 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. A formal definition of an annotated example and the learning algorithm can be found in [8].

An example is a list of instantiated actions (action type plus specific values for parameters) that constitute the achievement of a goal in the domain. For instance, the user may demonstrate how to operate the gas turbine engine using the simulator. This creates a minimally annotated example, where all actions are grouped under one non-primitive with a machine generated name. In Figure 2, the only change made by the user after the demonstration was to change the name of this top-level act to `StartEng`.[1]

Three specific annotations — $segmentation$, $unequals$, and *optional* — are described below. The PBD interface allows a user to make these (and other) annotations, as well as more mundane refinements such as adding or deleting steps.

Segmentation is an annotation that groups related actions; each such group is called a segment and is associated with a non-primitive act that is the purpose of the segment. Segments can be nested, so the elements in a segment can be

[1] In practice, annotating does not have to be a second pass — it can be interleaved with demonstrating.

Example 0

StartEng
1) PressGTGOn(Generator.GENERATOR)
2) OpenBleedValve(Engine.ONE)
3) OpenFuelValve1(Engine.ONE)
4) OpenFuelValve2(Engine.ONE)
5) PressGTEOn(Engine.ONE)
6) PressGTEEngage(Engine.ONE)
7) SetThrottle(Throttle.THROTTLE,Speed.FULL)

Figure 2: First example

Example 1

StartEngine
1) PressGTGOn(Generator.GENERATOR)
2) OpenFuelValves
1) OpenFuelValve2(Engine.TWO)
2) OpenFuelValve1(Engine.TWO)
3) OpenBleedValve(Engine.TWO)
4) PressGTEEngage(Engine.TWO)
5) SetThrottle(Throttle.THROTTLE,Speed.FULL)

Figure 3: Second example

either primitive acts or sub-segments. Segmentation is important because it directly influences the ability of the learning engine to correctly *align* the set of annotated examples, i.e. determine which portions of different examples are mapped to the same target concept. Unlike the other annotations, the user must provide segmentations since the learning techniques do not make any assumptions about how to segment an example. The process of segmenting an example can be a struggle for the user, since they must attempt to determine the best abstractions to represent intermediate goals.

Non-primitive acts have parameters, which are needed to propagate constraints between parameters of primitive acts in different sub-tasks of a task decomposition. Determining the number and type of parameters for non-primitives can be extremely difficult for a user even after an example has been segmented. Our generalization techniques make sound inferences about the number and type of parameters that non-primitive actions must have in order to be consistent with the set of examples.

Inferring non-primitive parameters assists the user; however, it can retard the convergence of the learning engine significantly. This is because coincidental equalities of parameter values for any two primitive actions will be propagated until another pair of acts, related by the same hierarchy of non-primitive acts, have different values for the parameter. The user can remedy this by adding *unequals* annotations for coincidental equalities.

A demonstration does not indicate whether a step is required either in general or for that particular demonstration. If desired, the user can explicitly indicate whether an action is optional through an annotation. In the absence of such annotations, inference is based on the set of pertinent examples (all examples that are aligned to the same target concept): if a step appears in every example, it is required; otherwise, it is optional.

Our learning algorithm distinguishes between positive evidence (e.g., annotating that a step is required) and the lack of negative evidence (e.g., the step appears in all examples aligned to this recipe). When given only segmentation annotations, inference is guaranteed to produce a task model consistent with all examples. However, additional annotations may lead to inconsistencies: e.g., if two segment elements are mapped to the same step but one is annotated as optional while the other is annotated as required.

A second partially annotated example is shown in Figure 3. In this example, a different engine was used, the user has decided to name the top-level act `StartEngine`, and the user has grouped two of the steps into a segment that has a purpose of type `OpenFuelValves`.

INTERACTING WITH GUESSERS

In typical PBD systems, it is solely the user's responsibility to specify any needed generalization knowledge. This section presents the framework for a system that lightens this burden through a set of guessers that suggest possible annotations; the user is free to ignore any or all suggestions. When learning task models, guessers can aid the user when segmenting, determining non-primitive parameters, marking optional steps, and making sure that aligned examples are consistent.

In our framework, a guesser is any algorithm that implements a task-specific application programming interface (API). For example, in our (Java-based) system that assists users to annotate demonstrations, two of the methods that a guesser must implement are:

```
public Boolean unequals (param1, param2)
public Boolean optional (step)
```

These methods allow the guesser to vote either "yes" (true), "no" (false), or "abstain" (null) about whether the guesser thinks a potential annotation should be suggested. This return value is typical, but some methods return a concrete value (e.g., for suggesting a new name of a segment). It would be a straightforward extension to allow for the guesser's response to include a confidence factor.

The following is a list of some knowledge sources that could constitute useful guessers (only example-based ones have been implemented). The descriptions are grounded by describing how a guesser of that type could simplify the user's efforts when annotating a demonstration.

Other examples A simple but effective guesser encapsulates another example defined by the user. Identifying sets of steps that constitute a nested segment can be done by looking to see if subsets of a segment in the current example constitute an entire segment in the previous example. When pieces of the current example and the previous example align to the same target concept, inconsistent annotations are readily identified.

The current generalization The generalization of the current examples can identify regularities that do not appear

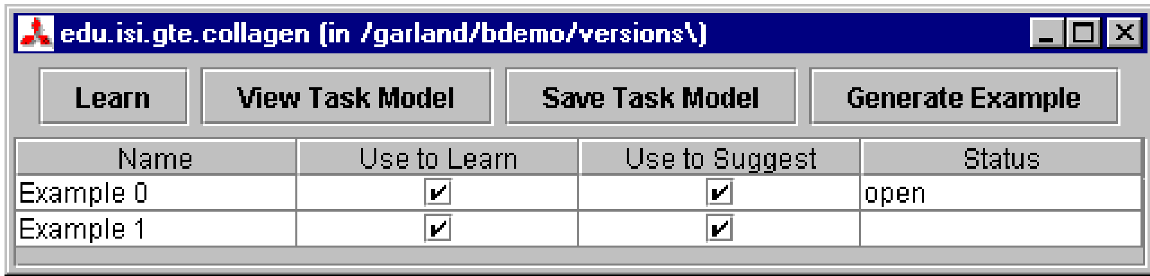

Figure 4: The user controls which guessers are active.

in any individual example. For example, perhaps there exists a recipe with four steps, two of which are optional. In that case, it is possible that different examples include each of the two optional steps, but none include both of them. None of these individual examples would recognize the group of four steps as a segment, but their generalization would.

The inference techniques In a similar vein to active learning (see the discussion section), awareness of the strengths and weaknesses of the generalization mechanism can lead to useful suggestions. For example, "distant" (i.e., separated by multiple non-primitive acts) equalities slow convergence when they are coincidental. Reflecting on this, the system could identify pairs of parameters for which it would be most useful to guess that the equality was coincidental.

Raw data Another source of suggestions could be a data mining algorithm operating on a pre-existing, unannotated usage log. Suggestions based on raw data can be very valuable since they may identify relationships the user was unaware of (but can verify) or had forgotten. For example, statistical techniques could be used to suggest that a step in a particular sequence is optional through analysis of many similar sequences. Such suggestion is most helpful if the step appears in all annotated examples.

Domain Theory In many PBD domains, there are relationships that are guaranteed to hold (or nearly so). In our case, recipes will rarely decompose into exactly n copies of a set of steps; instead, the recipe is likely to be recursive or be repeatable an arbitrary number of times (instead of for a fixed n). Such analysis may lead to segmentations that might not be made by an unaided user.

Heuristics Generalization can be viewed as search through a concept space, so many PBD inference problems will admit useful heuristics. In our case, a guesser might propose segmentations out of the space of all possible segmentations based on minimum description length criteria.

The user can control which guessers are active at any time. For example, guessers that encapsulate individual examples can be included or excluded by checking a box in the main task model development window (the "Use to Suggest" column Figure 4). The active guessers are organized into a committee (cf. Seung *et al.* [20]), which is run by a moderator. Two expected benefits are improved suggestions and less cognitive overhead for the user (since there is less information to manage).

In principle, there could be a different committee for each type of suggestion being voted on (e.g. segmentation, optionality, segment purpose name), but that has not been implemented. The committee moderator analyzes the current example in order to determine which suggestions should be voted on, and what constitutes a winning vote. The code for a given moderator can be non-trivial and is sensitive to the exact semantics associated with the guessers' API.

Our basic belief about the interaction style is that it should be controlled by the user. Thus, in our current implementation, suggestions are posted to a "bulletin board" window shown in Figure 5. The user can browse the suggestions there and double click on an entry to assert the suggestion (i.e., make the suggested transformation to the current example). This manipulation moves that suggestion into the undo stack and results in the computation of a new list of suggestions. Double clicking on the top of the undo stack retracts the suggestion (i.e., undoes the transformation).

In Figure 5, the user has returned to the first example to rename the top-level act to `StartEngine` to be consistent with the second example. After doing so, he sees two suggestions from Example 1: a good one for making a subsegment and a bad one for renaming the top-level act to be OpenFuelValves. Also, the user can easily restore the name of the segment to `StartEng`.

The interactions just described could be extended into a mixed-initiative collaboration. For example, the user could be involved in the voting process by overriding the criteria for a vote

Suggestions for Example 0

Type	Suggestion	Guesser
segmentation	[4#openfuelvalve1, 5#openfuelvalve2]--> subseg	[Example 1]
name change	StartEngine-->OpenFuelValves	[Example 1]

Type	Suggestion	Guesser
name change	StartEngine-->StartEng	Undo Manager

Figure 5: The table of suggested annotations.

to pass, based on the type of suggestion and the number of yes/no/abstain votes. Orthogonally, the user could authorize the system to take the initiative for "good" suggestions. For example, perhaps a user considers any segmentation suggestion that has a 100% vote with > 3 votes to be a good one. In that case, the system could either automatically assert the suggestion, pop up a dialog box proposing the suggestion, or highlight the suggestion in the bulletin board to attract the user's attention. Different "good" suggestions could be mapped to different actions. Whether giving the system more initiative in this way would improve the user experience is an open research question.

DISCUSSION

A principle of our implementation is that moderators should take into account whether the proposed annotation will change the generalization that the system is constructing. However, the larger issue at hand is how to incorporate knowledge of the generalization process into system-user interactions. The user should find these interactions comprehensible and helpful.

One way to incorporate knowledge of the generalization mechanism is to involve the user in the learning algorithm. For example, if the user can answer "yes" or "no" to queries about membership or equivalence regarding the concepts being formed (e.g., the class of optional steps), a system can learn in polynomial time [1]. Unfortunately, answering such queries can be difficult for the user.

In the *active learning* [5] paradigm, the system exerts some control over the input to the generalization mechanism. The standard approach along this line is for the learner to actively select from a known distribution of inputs to be classified by the user in lieu of relying on random sampling from that distribution. However, in programming by demonstration systems, the user — not the system or a random selection process — generates the examples. Thus, in this setting, an active learner is limited to selecting from among the possible pieces of generalization knowledge that might be relevant to the current example.

Two important issues to be dealt with by a PBD system are the possibility that the user will want to respond "I don't know" or may make classification errors (cf. Angluin *et al.* [2]). In our case, errors could be introduced by outright incorrect annotations or by the user reconceptualizing the domain. Namely, over the course of defining several examples, the user may change what they believe to be the correct answer. Reconceptualizing the domain can lead to the changes in segmentation and the name of the top-level act in Figures 2 and 3.

The only reason for moderators to make suggestions that will not change the output of generalization is to seek confirmation for conclusions that are weakly supported. For example, a user might "surprise" the system by indicating that a step is required despite evidence to the contrary. Such negative examples are often very valuable to learning techniques.

The intuition about seeking confirmation is formalized by probabilistic approximately correct (PAC) learning theory [22]. For example, if a step appears in $n - 1$ out of n pertinent examples, the step will be inferred to be optional, regardless of how large n is. In a PAC framework, however, the probability that this is correct decreases as n increases. If the probability of correctness drops below some specified threshold, the system can initiate an interaction. PAC techniques are robust in the face of classification errors.

Our system provides another tool to help the user to understand generalization. While depressing a button labeled "preview", the user sees how the generalization will "explain" the current example. For example, the visualization will indicate the inferred optionality of steps, inferred parameters for non-primitives, parameters that are bound to domain constants, and parameters that are constrained to be equal. This explanation is thus a way to show a portion of the generalization in a very grounded manner.

RELATED RESEARCH

Bauer *et al.* [3] developed PBD techniques for training intelligent agents to retrieve information from various web sites. A key part of the system is a training dialog between the learning agent and the system, where the agent makes suggestions for the next actions to be taken or the next landmark to be used. These suggestions are derived mainly based on the agent's understanding of the domain (i.e., HTML structure).

Gil and Melz [10] and Kim and Gil [13] 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.

Constable [4] shows the synergy that results from incorporating expectations from multiple knowledge sources during knowledge elicitation. In Constable, knowledge about background theories and knowledge about interdependency models are combined to support the acquisition of procedural knowledge. Constable includes a mechanism for making suggestions to the user about how to fix programming constructs that are ill-specified.

In a complementary approach, DIAManD [23] uses a committee of heuristics to support a user's interaction with an underlying PBD system. For example, DIAManD can run on top of SMARTedit, a PBD system that learns repetitive tasks in a text-editing domain. DIAManD's committee proposes *how* the user should add knowledge (for example, by

recording a full example or by stepping through the present formulation of a macro). In contrast, our committees propose knowledge to add to an example, but do not help a user choose between continuing to annotate this example and defining a new example.

CONCLUSION AND FUTURE WORK

This paper presented a PBD system that interacts with the user when defining examples that will be generalized. The user-initiated interactions are based on suggestions for possible annotations, which specify generalization knowledge about pieces of the example. The suggestions are proposed by committees of guessers; each guesser could represent an individual example, a generalization of examples, a data base of raw data, theory-based techniques, or arbitrary heuristics. A key principle for the committee moderator, which controls what suggestions are voted on, is to make suggestions that are sensible in light of the generalization mechanism of the systems. We have indicated how this approach helps to create segmentations, speeds sound learning of non-primitive parameters, and identifies inconsistencies between annotated examples.

Two key technical issues not yet fully addressed are ensuring the ongoing viability of proposed suggestions and controlling the timing of committees. As to the first point: in general, there is no guarantee that a suggestion made earlier will remain valid if the user changes the current example, either directly through the interface or through taking an alternate suggestion. Our current approach is to recompute the set of suggestions from scratch each time, but clearly incremental update methods would be preferable.

The second issue is that some individual guessers may take much more computation time than others, enough so that the user may be forced to wait for the system. Guessers should be run on their own execution threads so this can be avoided; however, there must be a way for the moderator to easily terminate this thread. First, related to point above, the thread may not complete before the user has changed the current example, thus invalidating the computation when it does terminate. Or, the moderator may be able to determine the outcome of the vote regardless of the value returned by guessers that are still computing. Our current implementation is synchronous.

ACKNOWLEDGEMENTS

I would like to thank Neal Lesh and Steve Wolfman for many thoughtful discussions and insightful comments about this work.

REFERENCES

1. D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.
2. D. Angluin, M. Kriķis, R. Sloan, and G. Turán. Malicious omissions and errors in answers to membership queries. *Machine Learning*, 28:211–255, 1997.
3. M. Bauer, D. Dengler, and G. Paul. Trainable information agents for the Web. In Lieberman [14], pages 87–114.
4. J. Blythe. Integrating Expectations from Different Sources to Help End Users Acquire Procedural Knowledge. In *Proc. 17th Int. Joint Conf. Artificial Intelligence*, pages 943–949, 2001.
5. D. Cohn, L. Atlas, and R. Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.
6. A. Cypher. Eager: Programming Repetitive Tasks by Example. In *Proc. ACM CHI '91*, pages 33–39, 1991.
7. A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.
8. 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.
9. A. Garland, K. Ryall, and C. Rich. Learning Hierarchical Task Models by Defining and Refining Examples. In *First Int. Conf. on Knowledge Capture*, 2001.
10. 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.
11. D. Halbert. SmallStar: Programming by Demonstration in the Desktop Metaphor. In Cypher [7], pages 102–123.
12. K. Kahn. Generalizing by Removing Detail. In Lieberman [14], pages 21–43.
13. J. Kim and Y. Gil. Aquiring problem-solving knowledge from end users: Putting interdependency models to the test. In *Proc. 17th Nat. Conf. AI*, pages 223–229, 2000.
14. H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.
15. D. Maulsby and I. Witten. Inducing programs in a direct manipulation environment. In *Proc. ACM CHI '89*, pages 57–62, 1989.
16. R. McDaniel and B. Myers. Getting more out of programming-by-demonstration. In *Proc. ACM CHI '99*, pages 442–449, 1999.
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. J. Rickel, N. Lesh, C. Rich, C. Sidner, and A. Gertner. Using a model of collaborative dialogue to teach procedural tasks. In *Proc. AIED Workshop on Tutorial Dialogue Systems*, 2001.
20. H. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proc. 5th Workshop on Computational Learning Theory*, pages 287–294, 1992.
21. D. Smith. PYGMALION: An Executable Electronic Blackboard. In Cypher [7], pages 19–48.
22. L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.
23. S. Wolfman, T. Lau, P. Domingos, and D. Weld. Mixed Initiative Interfaces for Learning Tasks: SMARTedit Talks Back. In *Proc. IUI-2001*, pages 167–174, 2001.