Acquiring Knowledge from Users in a Reflective Architecture

Yolanda Gil
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
gil@isi.edu

Abstract

As a system acquires more knowledge about a task it should be able to take a more active role in learning additional knowledge. To do so, it must be able to reflect upon the knowledge that it has—at each moment—and understand what every piece of knowledge, old and new, contributes to the task. This is the approach taken in EXPECT, a system that represents and manipulates many different types of knowledge distinctly and explicitly, and is able to determine what additional knowledge it needs to learn from a user based on its current knowledge about the task. This kind of active participation in the learning process is essential in learning complex tasks, not only from users but also from any other external sources of knowledge, be it information servers or other agents.

1 Introduction

Most machine learning approaches for acquiring knowledge from users reduce the user's participation in the learning process to a bare minimum. Learning apprentices, for example, acquire knowledge by analyzing a user's *normal* interaction with a system [Dent *et al.*, 1992; Mitchell *et al.*, 1990; Wilkins, 1990]. Some systems learn from solutions given by the user, while others display a proposed solution that the user can accept, correct, or rate [Tecuci, 1992; Baudin *et al.*, 1993; Dent *et al.*, 1992; Maes and Kozierok, 1993]. An advantage of these approaches is that they are non-intrusive, and users do not need to understand the underlying learning process beyond sometimes the fact that it is analyzing their examples. To learn under this limited style of interaction, the learning system is responsible for figuring out on its own knowledge about what features of the example are important, what new abstract features (not appearing in the examples) need to be defined, what combinations of those features are relevant to the task, and what parts of the example correspond to the user's preferences in that specific case. This approach is impractical in complex tasks where many different features in an example, many alternative values of these features, and many combinations thereof can account for user behavior. Theoretical analyses show the intractability of many learning tasks when the learner passively accepts examples, and at the same time point out the advantages of different kinds of active questions that a learner can ask of an external source of knowledge in terms of reducing the complexity of the learning task [Pitt and Valiant, 1988; Angluin, 1987; Ling, 1991].

An alternative and perhaps more practical approach is represented by learners that can request information from the user when faced with a situation where they do not know how to proceed [Huffman and Laird, 1993; Tecuci, 1992]. A drawback of this approach is that it puts more burden on users in that they are required to determine what information the learner needs and to figure out what is the best way to provide it, often by designing good examples that facilitate learning. As a system acquires more knowledge about a task, information about which knowledge the system already has and how it is organized may be unknown to the user and often hard to keep track of, posing increasingly stronger requirements on the user.

Ideally, learning systems should place less demands on the user and become increasingly more competent at the learning task as their knowledge bases grow. They should become more responsible for their internal organization of knowledge and of their own learning process, perhaps by asking more specific questions to the user or by pointing out areas in their knowledge base where they detect the need for additional information. In order to achieve this, they must be able to reflect upon the knowledge that they have about the task—at each moment—and understand what every piece of knowledge, old and new, contributes to the task. This understanding would allow them to be responsible for ensuring harmonious interactions between any new knowledge provided by the user and existing knowledge, thereby preventing redundancies, inconsistencies, and knowledge gaps that may arise inadvertently.

This is the approach taken in our work on EXPECT [Swartout and Gil, 1995; Gil, 1994; Gil and Paris, 1994]. EXPECT represents and manipulates many different types of knowledge distinctly and explicitly, and is able to derive and understand their functionality and their interactions. By representing knowledge declaratively, EXPECT has access to an understanding about the task that supports a more participative approach to acquiring knowledge from users. After describing briefly how EXPECT's knowledge acquisition tool figures out what kinds of knowledge it needs to acquire, we present two scenarios that illustrate EXPECT's interaction with the user.

2 Knowledge Acquisition in EXPECT

In EXPECT, any information necessary to perform a task is represented distinctly according to its nature either as domain facts or as problem-solving knowledge. Factual domain knowledge about a task (e.g., descriptions, relations, and definitions) is represented in LOOM [MacGregor, 1991], a state-of-the-art knowledge representation system of the KL-ONE family. Problem-solving knowledge is represented in a procedural-style language that allows subgoal posting, control programming constructs, and expressive parameter typing. EXPECT captures the semantics of goals by translating them into LOOM. EXPECT's problem solver uses this representation to reason about problem-solving goals and their relation to concepts and instances in the domain. More detailed descriptions of EXPECT's architecture and knowledge acquisition tool can be found in [Swartout and Gil, 1995; Gil, 1994; Gil and Paris, 1994; Swartout *et al.*, 1991].

Figure 1 shows an example of EXPECT's representation of factual and problem-solving knowledge in a transportation domain. The first expression specifies that seaport is a kind of port that has some ships, berths, covered storage area, and piers. It also indicates that there are two subtypes of seaport: inland waterway and maritime. The second expression is a very simple method that achieves the goal of finding the seaports of a location, returns several seaports, and does this by retrieving the value of the `r-seaports` relation of the location. The last expression is a more complex method that generates further subgoals.

Like many other systems that learn by doing, EXPECT can solve instantiated goals (like how to go to San Diego) and could use the resulting trace to learn from that specific example. Unlike other systems, EXPECT can be given generic goals (like how to go to any city) and use the resulting trace to understand what kinds of knowledge are needed for an entire class of tasks, i.e., for all possible instances of the given generic goal.

EXPECT's knowledge acquisition module is informed by the problem solver any time that errors in the knowledge bases are encountered. Errors (or simply anything that the system is not sure about and would like the user to check) may also be detected by the parser and by a module that analyzes individual methods and instances. Instead of interrupting problem solving when an error arises, the system takes it as a need for the knowledge acquisition module to request the user's intervention. The problem solver reasons about these errors and provides detailed information to the knowledge acquisition tool that is crucial to support the user in correcting them. Besides errors, the problem solver also signals possible problems that may result from a user's changes to the knowledge base. The user may not always foresee these possible problems, so the system takes the responsibility to track them. Sometimes these problems are not necessarily errors: often they reflect things that the system brings to the user's attention and can be dismissed by the user after giving them consideration.

EXPECT has a catalog of possible types of errors together with possible actions that may be suggested to the user in order to correct each type of error. The information about errors is explicitly represented, and we find it is very useful as a means for the problem solver to provide feedback to the knowledge acquisition tool regarding the status of the current knowledge base. Table 1 describes some of these errors.

Figure 2 shows a snapshot of EXPECT's user interface. It allows the user to examine the content of the knowledge bases and to navigate through problem-solving episodes to understand how the system achieves the task goals. The left side of the screen shows a problem-solving trace. The right side always shows the current errors as pending items on an agenda. We will come back to this figure since this is one of the situations discussed in our example scenarios.

3 Acquiring Factual Knowledge

Suppose that the user wants to add a new instance `Long Beach` of type `port`. When a new instance is defined, EXPECT checks that the type given is specific enough. In this case, the type is `port`, whose subtypes are `airport` and `seaport`. The problem solver's analysis of the task shows that the system transports objects to seaports and never to airports. This is an indication for the knowledge acquisition tool that it is important to know if `Long Beach` is a seaport. Thus, EXPECT requires the user to be more specific about the type of port that `Long Beach` is. The user is shown the subtypes of port and is asked to pick among them. If the user does not know this information, EXPECT will accept `port`, but will place an item on the agenda to remind the user to provide this information when it becomes available (notice that this corresponds to

```
(defconcept SEAPORT
  :is-primitive port
  :constraints (and (some r-ships ship)
                    (some r-berths berth)
                    (some r-covered-storage-area number)
                    (some r-piers number))
  :partitions ((inland-or-maritime-seaport (inland-waterway-seaport maritime-seaport))))

(defmethod FIND-SEAPORTS
  :goal        (find (obj ?s is (set-of (inst-of seaport)))
                     (of (?loc is (inst-of location))))
  :result      (set-of (inst-of seaport))
  :method-body (r-seaports ?loc))

(defmethod DETERMINE-WHETHER-SHIP-FITS-IN-SEAPORT
  :goal        (determine-whether
                     (obj (?ship is (inst-of ship)))
                     (fits-in (?port is (inst-of seaport))))
  :result      (inst-of boolean)
  :method-body (and (less (obj (r-length ?ship))
                          (than (r-length (r-berths ?port))))
                    (less (obj (r-draft ?ship))
                          (than (r-depth (r-berths ?port)))))
```

Figure 1: Factual and problem-solving knowledge in EXPECT.

errors and potential problems	*source*	*suggested corrections*
type of instance I is not specific enough for method M	instance analyzer	provide more specific type for instance modify method body delete instance
missing filler of role R of instance I needed in method M	instance analyzer	add information about instance modify method body delete instance
string S in method M does not have a syntactic category	parser	modify method add instance, concept or relation define action name or action role name
string S in method M belongs to two syntactic categories	parser	modify method add instance, concept or relation S define action name or action role name S
method M cannot be parsed	parser	modify method
method M has more than one parse	parser	modify method
method M is not used	method analyzer	modify method goal delete method
variable V declared in method M is not used in the method's body	method analyzer	modify method body modify method goal
variable V in step S of the body of method M is not declared in method M	method analyzer	modify method body modify method goal
result returned by a step S in the body of method M is not used	method analyzer	modify method body
no method found to achieve goal G in the body of method M	problem solver	modify method body modify another method's goal add a new method
role R undefined for type C in method M	problem solver	modify method M add relation R for type C

Table 1: Potential problems in the knowledge bases detected by EXPECT. Each component of EXPECT is able to detect an error, recover from the error if possible, and report the error's type and context in which it occurred. EXPECT's knowledge acquisition tool uses this information to support users in resolving errors.

Figure 2: EXPECT's knowledge acquisition interface.

the first entry in Table 1.) But let us suppose that the user picks `seaport` as the type of the `Long Beach` port. EXPECT checks again whether this type is specific enough. There are two subtypes of seaport (inland waterway and maritime), but neither one of these subtypes are used during problem solving. Thus, EXPECT does not request the user to be more specific.

Next, EXPECT checks what information is needed about seaports for problem solving. One method uses the ships available at a seaport, another uses the berths. So the system asks the user for this information. Again, if this information is not available, EXPECT will place these requests in the agenda.

Now suppose that the user adds a new instance called `Los Angeles` of type `location`. This type is specific enough for the problem-solving methods, and the information that they need about locations is what seaports they have. EXPECT includes a new item in the agenda to request this information. This is the situation shown in Figure 2. The smaller window in the lower right shows a description of `Los Angeles`, where the system indicates that it is a `location` and that additional information about the seaports of `Los Angeles` is needed in order to use this instance for problem solving. Notice that this request is also an item on the agenda (the first one).

At this point the agenda contains several items, each representing requests for information about the new instances just defined. EXPECT's support goes beyond detecting these problems. Because EXPECT understands the reasons for each error, it can support the user in resolving each item on the agenda. Consider the item requesting information about the seaports of `Los Angeles`. If the user selects this item, EXPECT provides an explanation of why this information is needed (i.e., which methods use this and what they accomplish). EXPECT also suggests possible actions that the user can undertake to resolve this agenda item based on what it knows about this kind of error. In this case, the item indicates a missing value of a role for an instance. EXPECT suggests that the user 1) provide a value, 2) remove the instance, or 3) modify the method so that this information will not be needed. EXPECT dynamically creates suggestions that are specific to the agenda item. In this case, if the user chooses to provide a value, EXPECT will present the user with a list of instances of `seaport`. Upon the choice of `Long Beach`, the agenda item disappears.

This knowledge acquisition dialogue illustrates several important points. First, the system *understands* how each type of instance is used and provides support for knowledge acquisition based on this understanding. If the methods change and new information about a type of instance is required, the system will realize this and update the agenda accordingly. Second, the user is *insulated from the details* of the implementation because EXPECT keeps track of what information is needed. Third, the agenda makes the dialogue very *flexible*, since the user chooses when to attend to an item raised by the system.

4 Acquiring Problem-Solving Knowledge

EXPECT also allows a user to modify problem-solving knowledge. The user can add more detail to an existing method by inserting steps in the method body, or change a method's goal by adding new parameters or modifying the types of the ones that it currently has.

If the user creates the method from scratch, EXPECT can provide little help because it will not have an understanding of this new method until the user is finished with it. Instead, EXPECT encourages the user to re-use existing methods in the knowledge base by using them as the basis for new methods. Because EXPECT already understands these methods, it can provide help in adapting them to new uses.

For example, suppose that the user wants to add a method to determine whether an aircraft fits in a hangar. The user indicates that the new method will be similar to the existing method to determine whether a ship fits in a seaport (shown in Figure 1). EXPECT uses the latter as a model, counting on the user to specify all differences. The user indicates that `ship` has to be changed to `aircraft`, and `seaport` to `hangar`.

EXPECT builds a new method with these changes and adds it to the problem-solving knowledge base. Then it checks how the new method affects the rest of the knowledge currently in the system. The first check concerns the validity of the new method in itself. This includes looking for syntax errors, inconsistent type passing through parameters, and steps in the method whose results are not used. Each error detected becomes an item on the agenda. The second matter that EXPECT checks is how the new method relates to the rest of the methods. EXPECT understands that methods are used to achieve goals, so it will run the static analyzer and make sure that the method is useful and that the subgoals that the method body contains can be achieved by other methods. Again, EXPECT warns the user via the agenda if it finds any problems. Last, EXPECT checks if the new method needs information about instances that is not currently available. The same checks are performed if a user modifies an existing method.

In our example, EXPECT will notice that `r-berths` is not a valid role of aircraft. For this type of error, EXPECT has two suggestions for the user: to modify the method, or to define that role for the concept. For the first error, the user chooses to modify the method and deletes that step. For the second error, the user chooses to modify the method so that instead of `r-draft` it uses `r-depth` of aircraft. In effect, EXPECT is providing support in completing the mapping and adaptation in this analogical process.

The scenario just described clearly follows an analogy process, but one where EXPECT shares the task with the user: the user suggests the source of the analogies, the mapping, and any necessary adaptations, while the tool provides the supporting environment for navigating through the system's reasoning and carrying out the user's corrections through analogical reasoning or any other mechanisms. It is important to point out that if the user leaves out anything that is relevant, the analysis that EXPECT performs to check the validity of the new method may detect that this is the case and raise agenda items to be resolved by the user.

5 Conclusion

We have presented EXPECT, a system that takes an active role in learning new knowledge from users. EXPECT derives a rich representation of the functionality of each piece of knowledge about a task. The knowledge acquisition tool uses this functionality to reason about knowledge interactions and to support the user in changing the knowledge base. EXPECT analyzes the consequences of each individual change that the user makes and cooperates with the user in the process of resolving them.

Our work in EXPECT shows that the better a system understands the task knowledge, the better equipped it is for interacting with users to acquire further knowledge. This kind of active participation in the learning process is useful not only in in learning from users but also from any other external source of information for that matter, be it information servers or other agents.

Acknowledgements

The author would like to thank current and previous members of the EXPECT group, in particular Bill Swartout, Marcelo Tallis, Pedro González, Bing Leng, Vibhu Mittal, Cécile Paris, and Ramesh Patil. Kevin Knight, Karl Schwamb, and Sheila Tejada provided many useful comments on a draft of this paper. We gratefully acknowledge the support of the Advanced Research Projects Agency under contract no. DABT63-91-C-0025. The view and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

References

Angluin, D. 1988. Queries and concept learning. *Machine Learning*, 2(4):319–342.

Baudin, C., Kedar, S., Underwood, J. G., and Baya, V. 1993. Question-based aquisition of conceptual indices for multimedia design documentation. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC.

Dent, L., Boticario, J., McDermott, J., Mitchell, T., and Zabowski, D. 1992. A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA.

Gil, Y. 1994. Knowledge refinement in a reflective architecture. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA.

Gil, Y. and Paris, C. 1994. Towards method-independent knowledge acquisition. *Knowledge acquisition*, 6(2):163–178.

Huffman, S. B. and Laird, J. E. 1993. Learning procedures from interactive natural language instructions. In *Proceedings of the Tenth International Conference on Machine Leaning*, Amherst, MA.

Ling, X. 1991. Inductive learning from good examples. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia.

MacGregor, R. 1991. The evolving technology of classification-based knowledge representation systems. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, ed. J. Sowa. San Mateo, CA: Morgan Kaufmann.

Maes, P. and Kozierok, R. 1993. Learning interface agents. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC.

Mitchell, T. M., Mahadevan, S. and Steinberg, L. 1990. LEAP: a learning apprentice for VLSI design. In *Machine Learning: An Artificial Intelligence Approach*, volume 3. Morgan Kaufmann, San Mateo, CA.

Pitt, L. and Valiant, L. G. 1988. Computational Limitations on Learning from Examples. *Journal of the Association for Computing Machinery* (35):965–984.

Swartout, W. R. and Gil, Y. 1995. EXPECT: Explicit Representations for Flexible Acquisition. *Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada.

Swartout, W. R., C. L. Paris, and J. D. Moore. 1991. Design for explainable expert systems. *IEEE Expert* 6(3):58–64.

Tecuci, G. D. 1992. Automating knowledge acquisition as extending, updating, and improving a knowledge base. *IEEE transactions on Systems, Man, and Cybernetics* 22(6):1444–1460.

Wilkins, D. C. 1990. Knowledge Base Refinement as Improving an Incorrect and Incomplete Domain Theory. In *Machine Learning: An Artificial Intelligence Approach, Vol III*, pages 493–513. Morgan Kaufmann, San Mateo, CA.