Knowledge Refinement in a Reflective Architecture

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

Abstract

A knowledge acquisition tool should provide a user with maximum guidance in extending and debugging a knowledge base, by preventing inconsistencies and knowledge gaps that may arise inadvertently. Most current acquisition tools are not very flexible in that they are built for a predetermined inference structure or problem-solving mechanism, and the guidance they provide is specific to that inference structure and hard-coded by their designer. This paper focuses on EXPECT, a reflective architecture that supports knowledge acquisition based on an explicit analysis of the structure of a knowledge-based system, rather than on a fixed set of acquisition guidelines. EXPECT's problem solver is tightly integrated with LOOM, a state-of-the-art knowledge representation system. Domain facts and goals are represented declaratively, and the problem solver keeps records of their functionality within the task domain. When the user corrects the system's knowledge, EXPECT tracks any possible implications of this change in the overall system and cooperates with the user to correct any potential problems that may arise. The key to the flexibility of this knowledge acquisition tool is that it adapts its guidance as the knowledge bases evolve in response to changes introduced by the user.

Introduction

The knowledge about a task is not a collection of isolated information packets but rather a carefully constructed web of facts, data, and procedures. The details of how knowledge is organized and how it interacts may be unknown to the user and often hard to keep track of. The key to knowledge acquisition is thus not in supporting the addition of more items to the collection, but in ensuring harmonious interactions between new and existing knowledge and preventing redundancies, inconsistencies, and knowledge gaps that may arise inadvertently.

Most tools for knowledge acquisition achieve this by having expectations about how each piece of knowledge fits in the overall system (Marcus & McDermott 1989; Musen 1989; Kahn, Nowlan, & McDermott 1985). For example, systems for classification tasks need knowledge for mapping inputs into classes. When the user enters a new class, their acquisition tools always expect to be given knowledge about how to assign an input to that new class. Having these expectations is very useful to support knowledge acquisition in that they allow the system to ensure that changes are introduced by the user in a harmonious way. However, since each tool is designed for a specific type of task, the expectations are hard-coded in the tool. This limits a tool's flexibility, since applications often do not conform exactly to the type of task that a tool was designed for and multiple problem-solving approaches may be required within a given application. In addition, it is hard to determine beforehand the type of problem-solving method that is needed for a new application.

The goal of the EXPECT project is to build tools for knowledge refinement that are both flexible and supportive to the user. EXPECT forms dynamic expectations based on the current knowledge of the system by understanding the content of the knowledge bases and their interactions. These expectations are not hard-coded in the knowledge acquisition tool; instead, they are explicitly created by the system when needed. The key to this goal is a *reflective* architecture, i.e., a system that has the ability to introspect and determine exactly what every piece of knowledge, old and new, contributes to the task.

EXPECT's architecture is reflective because it represents and manipulates many different types of knowledge distinctly and explicitly. Factual domain knowledge about a task (e.g., descriptions, relations, and definitions) is represented in LOOM (MacGregor 1988; 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 concepts. The problem solver uses this representation to reason about actions and their relation to concepts and instances in the domain. By representing each type of knowledge declaratively and supporting interrelationships between them, EXPECT has access to better understanding about the task than other architectures may have.

This paper begins with an overview of EXPECT's architecture and its knowledge acquisition tool. Next we describe how a user interacts with the knowledge acquisition tool to change the knowledge initially given to the system, taking

examples from an application that evaluates transportation plans. The paper continues with a discussion of how EXPECT relates to other relevant work on knowledge acquisition. Finally, we present our plans for future work and a conclusion.

The EXPECT Architecture

EXPECT builds on previous research on the Explainable Expert System (EES) project (Neches, Swartout, & Moore 1985; Swartout, Paris, & Moore 1991). EXPECT's architecture is designed to provide an understanding of how each piece of information in a knowledge-based system contributes to solving a task. This understanding comes from various features of the architecture that we describe briefly in this section.

Figure 1 shows an overview of EXPECT's architecture. 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. Domain facts are represented in LOOM (MacGregor 1988; 1991). LOOM provides a descriptive logic representation language and includes a classifier for inference. Problem-solving knowledge is expressed as EXPECT's methods. A method in EXPECT is an abstract and generic description of how a goal can be achieved including the goal, a method body that describes the procedure to achieve that goal, and the result that the method is expected to return. The goal of each method is also represented in LOOM, referring to the concepts and instances that appear in the parameter list.

Figure 2 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 with ships, berths, covered storage area, and piers. The second expression is a very simple method to find the seaports of a location by retrieving the value of the `r-seaports` relation of the location. The last expression in the figure is a more complex method to determine whether a ship fits in a seaport. Beside relations, the method body can contain subgoals which can be combined using control structures such as conditional and iteration statements. The methods shown here are domain-specific, but domain-independent generic methods can be expressed with this same language.

In order to ensure coherence among the various types of knowledge, the problem solver uses the factual and problem-solving knowledge sources to perform a static analysis of a given top-level goal, recording how each piece of knowledge contributes to the problem-solving process. EXPECT employs the reasoning capabilities of LOOM augmented with goal refinement and reformulation as follows. EXPECT's analysis is effectively a partial evaluation of the given top-level goal. If no method is found to achieve a posted goal, the goal is reformulated using the factual domain knowledge into a set of subgoals that can be achieved. For example, if there is no method to find the speed of a ship and three types of ships are described in the domain knowledge, the system will reformulate this goal into three subgoals and look for a method to find the speed of each of the three types of ship. This analysis provides the knowledge acquisition tool with an understanding of both the functionality and the nature of all the information used for the task.

Another important source of understanding is the tight integration of the problem solver with the LOOM classifier. In addition to classes and instances, EXPECT represents in LOOM all the goals that arise during problem solving and matches goals and methods based on their semantics using LOOM's classifier. EXPECT's matcher can find methods to achieve a posted goal taking into account the semantic definitions of their respective arguments. Any goal (i.e., a posted subgoal or a goal that a method achieves) is represented as a concept. For example, the LOOM definition of the goal achieved by the method `find-seaports` (shown in Figure 2) is:

```
(defconcept goal-concept--find-seaports
  :is (and goal-concept FIND
       (the OBJ
            (and instance-set seaport))
       (the OF
            (and instance-description location))))
```

The matcher finds the type of the most general bindings for variables that can be used to unify a posted goal and the goal that a method can achieve. The goal's arguments may be given in any order, because the matcher maps arguments according to their names.

In EXPECT, problem-solving errors also provide useful information for knowledge acquisition. Errors (or simply anything that the system is not sure about and would like the user to check) may come from the parser, the matcher, or the static analyzer. 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, as we show in the next section.

In summary, the facility to relate all the different sources of knowledge in the system and capture their influence in the system's behavior enables EXPECT's knowledge acquisition tool to support the user in changing the system's knowledge.

Knowledge Acquisition in EXPECT

EXPECT's knowledge acquisition module is invoked any time that errors in the knowledge bases are encountered during problem solving. 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. We consider them possible *lapses* on the part of the user, and they are added to an *agenda* of items that require user intervention. For example, if the user adds a new method that achieves the same goal as a method that already exists, the system detects this and notifies the user. Lapses 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. The system bases the agenda's requests on information that

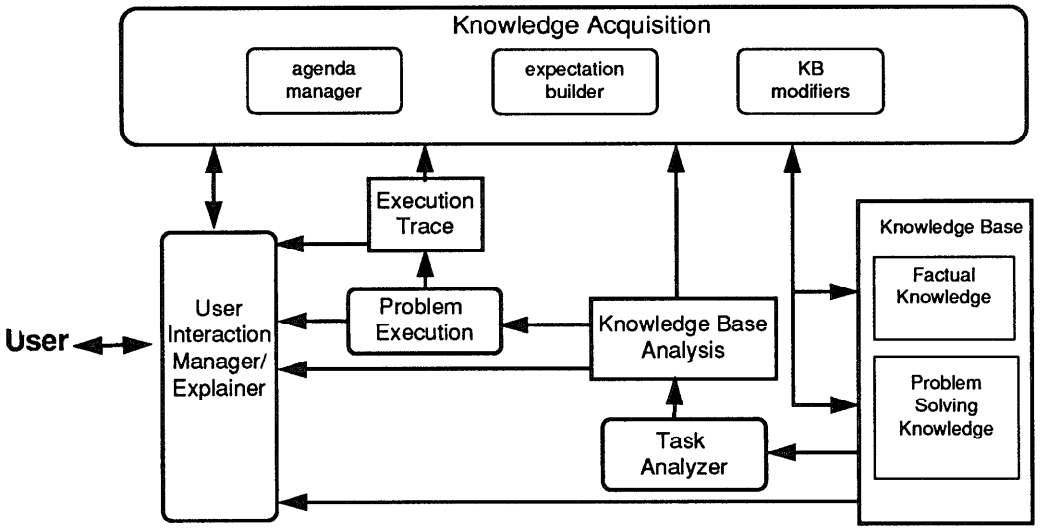

Figure 1: A schematic representation of EXPECT's architecture.

```
(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)))

(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  (less (obj (r-length ?ship))
                      (than (compute-max (obj (spec-of length))
                                         (of (set-of (spec-of berth)))
                                         (in ?port)))))
```

Figure 2: Factual and problem-solving knowledge in EXPECT.

is actually needed for problem solving, and as a result the user is not bothered with unnecessary interventions.

EXPECT has a catalog of possible types of lapses together with possible actions that the user may be suggested to take in order to correct each type of lapse. The information about lapses 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.

Figure 3 shows a snapshot of EXPECT's user interface. It allows the user to examine the content of the knowledge bases and navigate through problem-solving episodes to understand how the system achieves the task goals. The left side of the screen is showing the problem-solving tree and the right side the current items pending on the agenda. In this snapshot, the user had asked for a description of the instance `Los Angeles`, which caused the smaller window in the lower right corner to pop up. In the description of `Los Angeles`, 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. EXPECT knows that this is needed because it is used in the method to find the seaports of a location shown in Figure 2. Notice that this request is also an item on the agenda (the second one).

The user can interact with the system to resolve items on the agenda, or take the initiative to add new knowledge or change the knowledge already in the system. EXPECT analyzes how any change introduced by the user affects the problem solving required for the task, and tries to detect any negative consequences provoked by the change. These also raise lapses that are included in the user's agenda. The knowledge acquisition tool can be used to correct and extend the problem-solving knowledge as well as the factual knowledge. Table 1 summarizes the analysis that EXPECT performs for every modification done by the user. The next sections describe with examples how this analysis supports the knowledge acquisition process.

Adding New Instances

Suppose that the user wants to add a new instance. He or she selects the menu to add a new instance, and types:

Name: `Long Beach`
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 needs to know the ships available at a seaport. 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. But let us suppose that the user picks `seaport` as the type of the `Long Beach` port.

EXPECT next checks what information is needed for problem solving about this type of instance. One method uses the ships available at a seaport and its 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.

At this point the agenda contains several items, each representing requests for information about the new instances just defined. EXPECT provides the user with specific support for each type of item on the agenda. Consider the item requesting information about the seaports of `Los Angeles`. If the user clicks on this item, EXPECT pops up a menu with an explanation of why this information is needed (i.e., which methods use this and what they accomplish). The menu also contains possible actions that the user can undertake to resolve this agenda item. In this case, the item requests the value of a role for an instance. EXPECT suggests that the user:

- provide a value
- remove the instance
- modify the method, so it will not need this information.

Based on its underlying knowledge, EXPECT prescribes different solutions for each type of item in the agenda. This information is represented declaratively, and EXPECT uses it to dynamically create suggestions that are specific to the agenda item. In this case, if the user chooses to provide the value `Long Beach`, then the agenda item will disappear.

This knowledge acquisition dialogue contains 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.

Adding New 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. The user can also add new methods as we describe next.

If the user creates the method from scratch, EXPECT can provide little help because it will not have an understanding

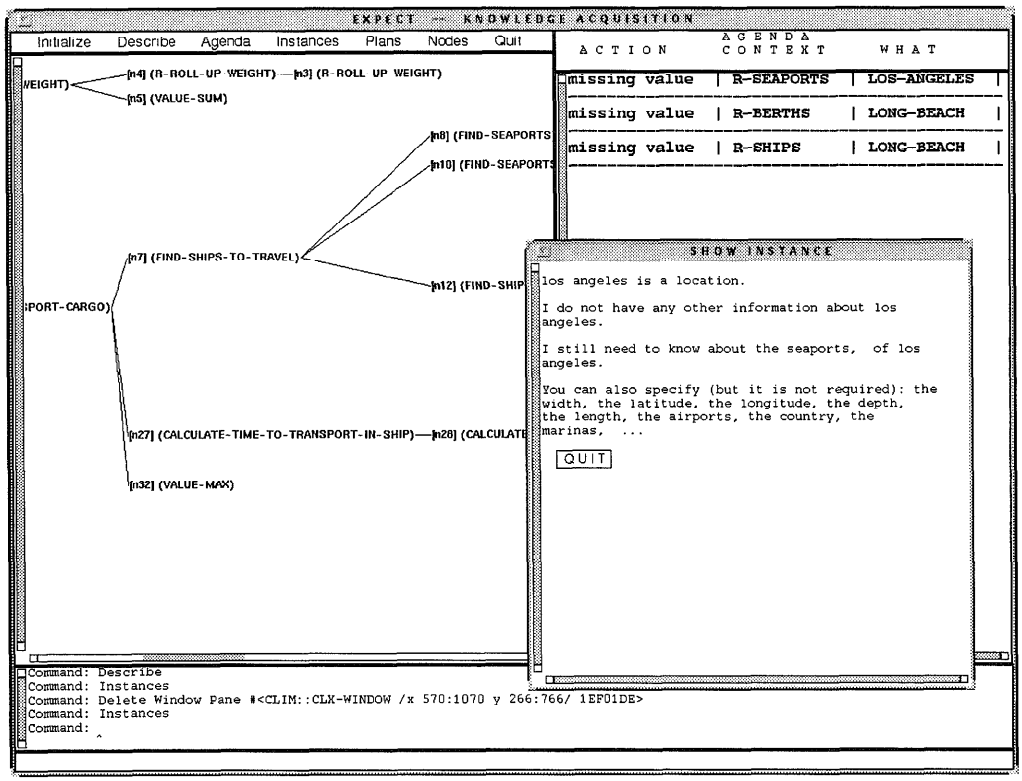

Figure 3: EXPECT's user interface.

When the user adds an instance of a type:

1. If the type given for the instance is more general than the types that are used for problem solving, find more specific types in the knowledge base and ask user to choose one.
2. Look up the roles defined for the instance type. Of those, find which roles are used for problem solving and ask the user for their value.

When the user changes any method:

1. If the new method uses a role of a type that it did not use before, ask user for the value of that role for all the instances of that type.
2. If the new method posts a subgoal, check that there is a method that matches with that subgoal. If not, notify the user.
3. If the new method is not used to achieve any subgoal, notify the user.
4. If the new method has syntactic errors, ask user for corrections.
5. If the new method contains information that is not used for problem solving, notify the user.

Table 1: EXPECT performs a thorough analysis of every modification done by the user.

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 find the airports of a location. In this case the user indicates that the new method to find airports is similar to the existing method to find seaports (the one shown in Figure 2). EXPECT uses the latter as a model, counting on the user to specify all differences. The user indicates that the relation `r-seaports` has to be changed by `r-airports`, and the concept `seaport` by `airport`.

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 same checks are performed if an existing method is modified.

The first check concerns the validity of the new method in itself. This includes looking for syntax errors, inconsistent type passing, and steps in the method whose results are not used. Each lapse 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. In this case, it realizes that `find-airports` retrieves the value of `r-airports` of instances of type `location` and consequently it adds an item to the agenda to request this value for `Los Angeles`.

The scenario just described clearly follows an analogy process. EXPECT provides a framework for analogical reasoning where the user suggests the source of the analogies, the mapping, and any necessary adaptations, and 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. We are currently extending our system to provide support for finding similar methods using a non-exact matcher as we discuss below. 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.

Related Work

EXPECT's reflective architecture provides an understanding of the knowledge in the system that can be used to form expectations about any new knowledge being added. Having expectations is be a powerful basis to support knowledge acquisition. TEIRESIAS (Davis 1980) used statistical techniques to form expectations about what terms were likely to co-occur. More recent tools, such as SALT (Marcus & McDermott 1989), PROTEGE (Musen 1989), and MORE (Kahn, Nowlan, & McDermott 1985), are built for a specific inference structure (e.g., classification) and expect their knowledge base to be populated with information useful for that type of task (Chandrasekaran 1986; McDermott 1988). However, since their expectations are hard-coded, these tools do not provide much flexibility (Musen 1992). The problem-solving structure of an application cannot always be defined in domain-independent terms. Furthermore, these method-specific inference mechanisms may not address some of the particulars of an application simply because they were designed with generality in mind. Another problem with the method-specific knowledge acquisition tools is that they raise the non-trivial issue of determining a library of possible methods. The work involved in handcrafting such a library of methods, making sure to both provide wide-coverage of tasks and well-understood characterizations of the inference capabilities of each method, is daunting.

To address these limitations, some researchers (Klinker *et al.* 1991; Puerta *et al.* 1992) are developing libraries of problem-solving methods that handle finer-grained inference structures than the ones above. These approaches provide more flexibility in building a knowledge-based system, and we share their belief that this is a step in the right direction. EXPECT's expectations are as fine-grained as the user's definitions. They are not hard coded, and are based on understanding each piece of knowledge both individually and in conjunction with others. EXPECT can be applied to tasks with any kind of inference structure.

NEODISCIPLE (Tecuci 1992) integrates several machine learning techniques in a knowledge acquisition tool. NEODISCIPLE takes a user-given answer to a problem and applies explanation-based learning to build a plausible proof tree, abduction to complete the proof, and several other learning techniques to generalize the proof. Its predecessor, DISCIPLE (Tecuci & Kodratoff 1990), built an analogy with an existing proof when the system lacked domain knowledge to build the proof for a new input. Our approach automates different parts of the analogical process. The user suggests the source of the analogies, the mapping, and any necessary adaptations. Our tool provides support for carrying it out, checking the validity of the new knowledge, and examining its effects in the current knowledge bases.

Discussion

We plan to extend EXPECT's reflectiveness in two main directions. One is to improve the understanding of goals through a relaxed semantic matcher, and the other is to extend the current representation of agenda items to provide more comprehensive support for the knowledge acquisition tool.

We have extended EXPECT's semantic matcher to find non-exact matches of methods and goals. By dropping parts of the definition of the goals, this relaxed matcher effectively does a partial unification. The relaxed matcher may propose a method that has five parameters that match exactly five of the six parameters of a posted goal. If a goal's parameter is of

a certain type and no methods are found that match exactly, the relaxed matcher may propose a method that applies to a more specific type or to another subtype of the same direct supertype. The relaxed matcher can also describe what relaxations yielded the retrieved method. We plan to use this relaxed matcher in our knowledge acquisition tool for several purposes. One is to suggest model methods when the user creates a new method (as in the `find-airports` example). Another possible use is in suggesting concrete solutions to agenda errors. For example, if no method is found to achieve a goal the system may suggest to use a method found by the relaxed matcher and indicate how to reduce their differences.

When EXPECT checks the effects in the knowledge bases of any change introduced by the user, the agenda reflects any possible lapses that may arise. Each type of item on the agenda is associated with a set of possible remedies that the system can suggest to the user to resolve the item. All of this information is explicit in the knowledge acquisition tool. We plan to categorize in more detail the possible lapses that may occur during knowledge acquisition, their relevance to the task at hand, and the possible actions to resolve them. For example, an item that signals that no method is available to achieve a frequently occurring goal is crucial, while an item requiring information on a location that is never used may be dismissed by the user. This categorization would allow the user to identify coherent states of the knowledge base during the knowledge acquisition process, when the system is ready for solving the task and can solve problems with an improved version of the knowledge base. Lapse categorization would also allow better management of the agenda through a priority mechanism based on the relevance of agenda items.

Conclusion

We have presented EXPECT, a reflective architecture for knowledge refinement that 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 support the user in changing the knowledge base. This makes EXPECT independent of the problem-solving method of the task, a key feature that distinguishes our approach from current knowledge acquisition tools.

Acknowledgments

The author would like to thank current and previous members of the EXPECT group, in particular Pedro Gonzalez, Bing Leng, Vibhu Mittal, Cécile Paris, Ramesh Patil, Bill Swartout, and Marcelo Tallis. The clarity of this paper was improved thanks to comments from Eduard Hovy, Kevin Knight, Bill Swartout, and the anonymous reviewers. 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

Chandrasekaran, B. 1986. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert* 1(3):23–30.

Davis, R. 1980. *Knowledge-based systems in artificial intelligence*. New York, NY: McGraw-Hill.

Kahn, G.; Nowlan, S.; and McDermott, J. 1985. Strategies for Knowledge Acquistion. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7(5):511–522.

Klinker, G.; Bhola, C.; Dallemagne, G.; Marques, D.; and McDermott, J. 1991. Usable and reusable programming constructs. *Knowledge Acquisition* 3(2):117–135.

MacGregor, R. 1988. A deductive pattern matcher. In *Proceedings of the 1988 Conference on Artificial Intelligence*.

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

Marcus, S., and McDermott, J. 1989. SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence* 39(1):1–37.

McDermott, J. 1988. Preliminary steps towards a taxonomy of problem-solving methods. In Marcus, S., ed., *Automating Knowledge Acquisition for Knowledge-Based Systems*. Boston, MA: Kluwer Academic Publishers.

Musen, M. A. 1989. Automated support for building and extending expert models. *Machine Learning* 4(3/4):347–375.

Musen, M. A. 1992. Overcoming the limitations of role-limiting methods. *Knowledge Acquisition* 4(2):165–170.

Neches, R.; Swartout, W. R.; and Moore, J. D. 1985. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering* SE-11(11):1337–1351.

Puerta, A. R.; Egar, J. W.; Tu, S. W.; and Musen, M. A. 1992. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition* 4(2):171–196.

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

Tecuci, G., and Kodratoff, Y. 1990. Apprenticeship learning in imperfect domain theories. In *Machine Learning: An Artificial Intelligence Approach*, volume 3. San Mateo, CA: Morgan Kaufmann.

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.