A Script-Based Approach to Modifying Knowledge Bases

Yolanda Gil and Marcelo Tallis
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
gil@isi.edu, tallis@isi.edu

Abstract

Our goal is to build knowledge acquisition tools that support users in modifying knowledge-based systems. These modifications may require several individual changes to various components of the knowledge base, which need to be carefully coordinated to prevent users from leaving the knowledge-based system in an unusable state. This paper describes an approach to building knowledge acquisition tools which capture knowledge about commonly occurring modification sequences and support users in completing the modifications they start. These sequences, which we call *KA Scripts*, relate individual changes and the effects that they have on the knowledge base. We discuss our experience in designing and compiling a library of KA Scripts. We also describe the implementation of a tool that uses them and our preliminary evaluations that demonstrate their usability.

Introduction

The maintenance of knowledge-based systems remains a largely unresolved problem after more than twenty years of research and practical experience in building knowledge-based systems. After initial prototypes are developed, subsequent modifications are usually made to detail and extend the knowledge base. Once the system is fielded, it would be extremely rare if the knowledge-based system did not need to be maintained to adapt to the changes that naturally happen in the world in which it works or to new requirements from its users. The problem of modifying a knowledge-based system is arbitrarily hard. Some modifications may involve complex restructuring or the introduction of large new portions that may be equivalent to the effort of building a whole new knowledge base. Providing support for these kinds of modifications will be very hard. But we should still be able to support modifications that change some aspect of the reasoning or add new simple steps.

Research in the area of knowledge acquisition only partially addresses the issue. Some tools allow users to populate a knowledge base with domain knowledge (Marcus and McDermott 1989; Eriksson *et al.* 1995; Puerta *et al.* 1992; Runkel & Birmingham 1993). The kinds of changes that a user can make is limited to filling in the knowledge roles determined by a predefined problem-solving method that the system uses. For example, in a configuration system the user could define new components but would not be able to change the configuration method to prefer certain configurations (e.g., cheaper ones). More automated approaches for building knowledge-based systems use machine learning and theory revision techniques (Langley & Simon 1995; Pazzani & Brunk 1991; Ourston & Mooney 1994). However, they can only be used for some types of problems (e.g., classification tasks). Other systems (Murray 1996) can assist users in fixing the inconsistencies caused by the addition of the new knowledge to a knowledge base, but without a problem-solving context in which the knowledge is used. Modifying a knowledge-based system requires a coherent sequence of several individual changes to definitions, facts, and methods that together compose the system. There is a need for tools that support users in coordinating these changes and carrying them out correctly.

We begin by describing the difficulties involved in supporting users as they modify knowledge-based systems. Then we discuss our approach and our initial implementation of a script-based tool that supports users in modifying a knowledge-based system. Our scripts represent typical sequences of changes that users can apply in order to complete modifications. Finally, we present the results of an evaluation that we conducted with several users and discuss the value of this approach and our plans for future work. Although this research is tied to our work within a particular framework for building knowledge-based systems, the problems addressed are described with enough generality that other researchers can benefit from our work. The paper also describes the specific features of this framework that we found useful to support script-based ap-

proaches to knowledge acquisition.

Why Modifying Knowledge Bases is Hard

Consider an example from our experience with a knowledge-based system for transportation planning. Suppose that the system calculates durations of trips involving only ships, and that now it has to be extended to consider aircraft too. This modification to the knowledge base involves several individual changes. First, existing knowledge may need to be modified. The description of vehicles (which may be represented as a concept with attributes or roles) has to be extended to include aircraft in addition to ships. The procedures (which may be represented for example with rules) to calculate round trip time need to be changed to take into account aircraft. New knowledge may also need to be added. For example, new procedures to calculate the round trip time of aircraft need to be added. In all these modifications, any new knowledge needs to be integrated with existing knowledge. The distance traveled is used in the new procedure for the round trip time of aircraft and it is also used in the already existing calculation for the round trip time of ships, so we need to make sure that they use consistent estimates of the distance.

Notice that if the user makes only some of these changes the knowledge base will be left in an *incoherent* state that will render it unusable because the system will not be able to solve problems. For example, suppose that the description of vehicles is extended to include aircraft but that no procedures to calculate the round trip time of aircraft are added. The system will no longer be able to estimate the duration of trips because it could not compute the round trip time of its aircraft. Because several changes are required to different pieces of the knowledge base, users can easily overlook some part of the overall modification and end up with an incoherent knowledge base. There are several reasons why it is hard for a user to complete the modification:

- **Separate pieces of knowledge:** The knowledge base is composed of many individual pieces of knowledge that come together during the reasoning, and it is hard to follow up on all of their interactions. We cannot use the unloading time of an aircraft in a procedure if we have not added a definition of what it is and specified its value for the different aircraft types.
- **Maintaining compatibility of types:** The arguments of expressions and the types of their results need to be compatible with how they are invoked and used. If speeds are specified in miles per hour then stopover times cannot be defined as a number because unless the system knows what units we used to measure these times it will not be able to add these quantities together correctly.
- **Automatic inferences not directly observable:** The interactions occur in the results of system-made inferences which are not directly observable to the user, such as class inheritance.
- **Propagation of interdependencies:** Modifications to a piece of knowledge may affect other components of the knowledge base and as a result require additional modifications. Furthermore, each of these additional modifications can in turn originate the need for additional changes. It is hard for a user to track down and to keep in mind all the modifications that are pending.

In sum, modifying a knowledge-based system often requires several individual changes to various individual components of different nature that need to be carefully coordinated. A good starting point is to build knowledge acquisition tools that find problems with the knowledge base and alert users about them, and in fact, this is pretty much the kind of support that a conventional compiler provides to programmers when they change their code. But helping users notice the problems only partially addresses the issue. Ideally, our tools should also support the user in resolving these problems by making suggestions about what additional changes may be needed in the knowledge base. To do so, the tool needs to have more context and some knowledge about the task that the user is trying to accomplish.

Our Approach: KA Scripts

Our approach is to equip knowledge acquisition tools with scripts that group many individual changes to represent how overall modifications are accomplished. A *Knowledge Acquisition Script* (or *KA Script*) is a prototypical sequence of changes together with the conditions that make it relevant given the previous changes to a knowledge base. An example of a KA Script is the creation of a new procedure that is similar to an existing one. It could be used to create the procedure that computes the round trip time of an aircraft based on the one for ships. The role of the knowledge acquisition tool is to help the user to resolve side-effects of changes already made and complete the modification that he or she has started. To provide this kind of support, a KA tool needs to have access to the following kinds of information:

- **problems with the current knowledge base**, which are indicative of what additional knowledge needs to be acquired from the user. For example the fact that the system is unable to calculate the round trip time of an aircraft indicates that the tool needs to acquire some procedural knowledge to calculate it. These problems are side-effects of previous changes. Possible problems with the knowledge base include errors (something it knows about is wrong) and knowledge gaps (something is missing). There may also be potential problems that need to be brought to the user's attention. We will refer to

all these as *errors* throughout the rest of the paper. The tool needs to be able not only to detect errors, but also to identify the problem-solving context in which they arise.

- **a history of the changes made to the knowledge base** to understand what the user has been trying to accomplish with the modification. If the tool is aware that the user has just changed a procedure to add two new calculations and there are no existing procedures to calculate them, then it can have the expectation that the user will define these procedures (and vice versa). Without this knowledge of the user's past changes the system would have a very myopic view of what is happening and suggest to the user to change the procedure back to the way it was. This would certainly accomplish the goal of taking the knowledge base back to a coherent state, but would not help the user make progress towards his or her goals.
- **a record of past versions of the knowledge base** to understand how the individual pieces of knowledge are supposed to come together. Suppose that the user initiates a modification that changes the arguments of some procedure and as a result the system cannot invoke it any longer. The system can use the past versions of the knowledge base to figure out which other procedures need to invoke it and how, and use this to help the user in completing the modification by updating those procedures.

For each problem in the knowledge base there may be several KA Scripts capable of fixing it. The knowledge acquisition tool can suggest which KA Scripts can be applied, but only the user has enough knowledge to decide which one is appropriate given the modification that he or she has in mind.

Figure 1 shows one of our KA Scripts, which we will explain in more detail in the next section. The error specified in a KA Script is a kind of problem that can appear in the knowledge base and makes the KA Script applicable. These errors can be detected automatically by analyzing whether the system can solve a specific task (e.g., calculating roundtrip time). The applicability conditions describe conditions (other than the error) that make the KA Script relevant to the situation. A short description and an explanation are used to show users what the KA Script will do if they choose to use it, and why it is being suggested by the system.

In designing KA Scripts, we took into account the following requirements to address their usability:

- They need to be at the right level of generality in the advice provided. A suggestion such as "Consider creating a new procedure for achieving the unmatched goal" is like to be less useful to the user than "Consider generalizing the procedure to calculate the round trip time for ships so it can be used for all vehicles". This does not mean that the KA Script needs to be described in great detail and in fact they

KA Script to resolve error type "Goal G-new cannot be matched"

Applicable when:
(a) A change has caused an argument A of a goal G to become more general, resulting in goal G-new
(b) Goal G was achieved by method M before A changed
(c) G-new can be decomposed into disjunctive subgoals G1 G2
(d) G1 is the same as G

Modification sequence:
CHOICE 1: Create new method M-new based on existing method
(1) System proposes M as the existing method to be used as a basis. User chooses M or another method.
(2) System proposes a draft version of M-new that modifies A to match G2. User can make any additional substitutions needed in the body of M-new.
(3) User edits body of M-new if modifications other than substitutions are needed.
CHOICE 2: Create new method M-new from scratch

Description of what this KA Script does:
Create a method that achieves goal G2 based on method M

Reasons why it is relevant to the current situation:
Method M was used before to achieve goal G, which was generalized to become the unmatched goal G-new. Now M can be used to achieve one of the subgoals in the decomposition of G-new. M may be used to create a new method that achieves the other subgoal in this decomposition.

Figure 1: A KA Script from our library.

are often more understandable when described in an abstract way.

- They need to be integrated with some basic knowledge acquisition tool and degrade gracefully, so that if no KA Script applies to the current situation (or the user does not want to use any of the applicable ones) then the user can still continue making changes. Since it is unlikely that we can design KA Scripts for all possible strategies that users can follow in changing a knowledge base, it would be too restrictive to force users to use KA Scripts only.
- They need to be structured to prioritize errors and to sequence pending changes in a way that makes the user's job easier. The errors can be prioritized because the process of fixing one error will often fix other errors. The changes need to be ordered so that they are presented to the user in a logical sequence that is easy to follow and understand, instead of jumping around several fixes, which can interrupt the flow continuously and make the user lose the thread of what he or she was doing. We also noticed that often the way a user makes a change sheds some light on how he or she may go about making other changes. So it is preferable to place earlier any changes that can be analyzed to guide subsequent changes. At the same time, many temporary errors can appear during a modification sequence and it is preferable that the user follows the chosen KA Script and does not interrup it to fix another error.

To create our library of KA Scripts, we first did a thorough analysis of the kinds of general changes and types of errors that could arise in using our baseline

knowledge acquisition tool. This analysis was done systematically by evaluating the effects of modifying every constituent in the grammar used to represent knowledge in our framework. The result of this analysis was the identification of all the possible error types and a set of KA Scripts for fixing them. These KA Scripts cover all the situations in which a user can get when modifying a knowledge base. However, our initial implementation showed that the guidance they provide to users is very vague. The main problem is that they are very general and as a result they do not make good use of the context available like previous modifications to the knowledge base or of its specific contents.

We then analyzed several hypothetical (yet plausible) scenarios for modifying a knowledge base. We looked at the changes that needed to be made, the errors that resulted from them, and how subsequent changes repaired the errors caused by earlier changes. We analyzed what kinds of advice would have been useful to users at each point, and determined what information from the context was needed to generate the advice automatically. The result of this effort was a set of KA Scripts that, though incomplete, were more specific to the context and as a result provided more help to a user.

Finally, KA Scripts produced by both methods were combined in a a single library. There is always some general KA Script in the library that applies to any situation. In situations where there is a more specific KA Script that applies, the guidance provided will be more specific to the context and more helpful. If not, we can fall-back on the general KA Scripts because they cover all situations and provide more generic (but still helpful) guidance. The result of this effort is a library of 75 KA Scripts that altogether address 23 types of errors in the knowledge base.

Several interesting issues came up in constructing the KA Script library. Initially we tried to organize KA Scripts by triggering them by user changes instead of errors. This produced scripts that were very cumbersome, mostly because potentially any other script was applicable at many points and it was hard to find a reasonable subset. Another approach that was not successful was to invoke scripts within a script. This produced a high degree of nesting and it would have been hard for users to follow what was happening. We also realized that KA Scripts could not only remind users of the changes that remain to be done but be useful checklists to help them keep track of the changes that they had already done.

KA Scripts are designed to be invoked after a user has performed some initial changes to the knowledge base. However, if the user makes an arbitrary number of changes and then turns to KA Scripts, it is hard to figure out how all the changes relate and provide helpful guidance. We assume a paradigm where the system starts with a coherent knowledge base (one that has no errors and can be used for problem solving), then the user makes a few changes (ideally just one) and invokes the tool, which uses KA Scripts to help the user bring the knowledge base back to a coherent state. Using an analogy with databases, we can view the process of modifying the knowledge base as a sequence of *transactions*, where KA Scripts support users by enforcing that transactions are completed so that the knowledge base is not left incoherent.

ETM: EXPECT's Transaction Manager

Our implementation of a script-based knowledge acquisition tool is **ETM** (EXPECT's Transaction Manager), a tool integrated with the EXPECT architecture for knowledge acquisition. We introduce some aspects of EXPECT as we present an example knowledge base and how ETM uses KA Scripts to guide users in modifying it. More details about EXPECT can be found in (Gil & Melz 1996; Swartout & Gil 1995; Gil 1994; Gil & Paris 1994).

EXPECT's knowledge bases contain factual domain knowledge and problem solving knowledge. The factual domain knowledge represents concepts, instances, relations, and the constraints among them. It is represented in Loom (MacGregor 1991), a knowledge representation system of the KL-ONE family. Problem solving methods are procedural descriptions for achieving goals. They consist of 1) a *capability* that represents the goal that the method can achieve, expressed with an action name and several parameters, 2) a *method body* that describes the procedure for achieving the method goal in the capability, and 3) a *result type* that specifies the type returned after elaborating the method body. Figure 2 shows examples from a simplified transportation domain. A vehicle is defined as a kind of major equipment that has a speed and can be either a ship or an aircraft. The method M2 specifies that in order to calculate the duration of a trip by ship from a location to another location we have to find the sailing distance between the locations and divide it by the speed of the ship.

EXPECT can be given general goals, such as `(calculate (obj (spec-of TRIP-DURATION)) (of (inst-of TRANSPORTATION-MOVEMENT)))`. General goals represent the kinds of goals that the system will be given for execution. EXPECT analyzes how to achieve these goals with the available knowledge. EXPECT expands a goal by matching it with a method and then expanding the subgoals in the method body. This process is iterated for each of the subgoals and is recorded as a derivation tree. Throughout this process, EXPECT propagates the types of the arguments performing an elaborate form of partial evaluation supported by Loom's reasoning capabilities. Using the derivation tree, EXPECT finds the interdependencies between the domain facts and the problem-solving methods, which are used by the knowledge acquisition tool to detect errors or knowledge gaps in the knowledge base and guide the user in resolving them. For example,

```
(defconcept VEHICLE
  :is-primitive (:and MAJOR-EQUIPMENT
                      (:the HAS-SPEED SPEED)
  :disjoint-covering (SHIP AIRCRAFT))

(defconcept TRANSPORTATION-MOVEMENT
  :is-primitive (:and TRANSPORTATION-DOMAIN-CONCEPT
                      (:the HAS-ORIGIN LOCATION)
                      (:the HAS-DESTINATION LOCATION)
                      (:the HAS-VOLUME-TO-MOVE TONS)
                      (:some HAS-AVAILABLE-LIFT SHIP)))

(def-expect-method M1
 (capability (calculate (obj (?t is (spec-of TRIP-DURATION)))
          (of (?m is (inst-of TRANSPORTATION-MOVEMENT)))))
 (result-type (inst-of ELAPSED-TIME))
 (body (pick (obj (spec-of MAXIMUM))
          (of  (calculate (obj  ?t)
                          (by   (HAS-AVAILABLE-LIFT ?m))
                          (from (HAS-ORIGIN ?m))
                          (to   (HAS-DESTINATION ?m)))))))

(def-expect-method M2
 (capability (calculate (obj  (?t is (spec-of TRIP-DURATION)))
                        (by   (?s is (inst-of SHIP)))
                        (from (?l1 is (inst-of LOCATION)))
                        (to   (?l2 is (inst-of LOCATION)))))
 (result-type (inst-of ELAPSED-TIME))
 (body (divide (obj (find (obj  (spec-of SAILING-DISTANCE))
                          (from ?l1)
                          (to   ?l2)))
               (by  (HAS-SPEED ?s)))))

(def-expect-method M3
 (capability (find (obj  (?d is (spec-of SAILING-DISTANCE)))
                   (from (?l1 is (inst-of LOCATION)))
                   (to   (?l2 is (inst-of LOCATION)))))
 (result-type (inst-of LENGTH))
 (body (if (or (unknown (obj ?l1)) (unknown (obj ?l2)))
       then (ask-user (obj SAILING-DISTANCE) (from ?l1) (to ?l2))
        else (look-up (obj (append ?l1 ?l2))
                      (in SAILING-DISTANCES-TABLE)))))
```

Figure 2: Some definitions of concepts and problem solving methods in a simplified transportation domain.

the derivation tree will annotate that in expanding M2 the speed of a ship is used. If a new ship is entered in the knowledge base and its speed is unknown, this will cause an error and the knowledge acquisition tool will ask the user to specify the speed. Other kinds of errors include goals that cannot be matched by any method, undefined parameter types, and method result types that are incompatible with what the method expansion actually returns.

Now suppose that the knowledge base in Figure 2 needs to be modified because the lift available for transportation movements is no longer only ships but can be any kind of vehicle. The available lift of a transportation movement needs to be changed from `SHIP` to `VEHICLE`. This causes an error because some instantiations of the `calculate` subgoal of M1 have no method matching them (the second parameter has changed to be of type `VEHICLE`). The user then defines a new method M2-PRIME based on M2 by substituting `SHIP` by `AIRCRAFT` and `SAILING-DISTANCE` by `FLYING-DISTANCE` and then adding the subgoal `calculate` to the method body to calculate the time spent in stopovers. These

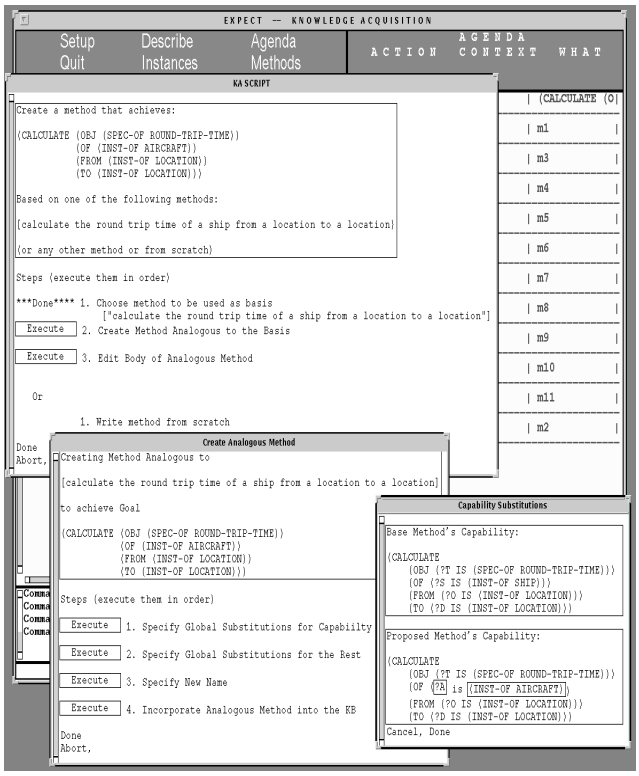

Figure 3: ETM's User Interface.

modifications now cause two additional errors: that the `find` and `calculate` subgoals of M2-PRIME cannot be matched. The user defines a new method M3-PRIME based on M3 to resolve the former, and writes a new method M4 for the latter. In summary, this overall modification required five individual changes to different parts of the knowledge base.

The KA Script shown in the Figure 1 is relevant when the first error arises in our example scenario. In this case, the goal to calculate a trip duration by ship (G) was generalized to calculate the duration by a vehicle (G-new). This new goal can be decomposed into the disjunctive subgoals calculate the duration for a ship and calculate the duration for an aircraft (G1 and G2, where G1 is the same as G). Since M2 was the method used to match the original goal, the KA Script proposes to create a new method based on M2. Figure 3 shows ETM's user interface when executing this KA Script. The current implementation has ten of the KA Scripts in our library, enough to support the test scenarios described in the next section. We are extending the system to include additional ones.

An important issue is the coordination of the execution of KA Scripts. We use a collaborative framework, where ETM finds the errors in the knowledge base and the KA Scripts that are relevant and the user decides which KA Script is most appropriate for the modification he or she has in mind. The overall control loop

for the execution of KA Scripts in ETM is as follows:

```
User makes change(s) in the knowledge base
ETM identifies errors in the knowledge base
While there are errors in the knowledge base
      ETM picks error e to be fixed and generates
            set K of KA Script candidates k that can fix e
      If the user does not choose any k
        then user can quit ETM and fix e with EXPECT,
              ETM can be invoked again anytime
        else user chooses one k from the set K,
              ETM helps user to apply k
      ETM identifies errors in the knowledge base
```

Detecting which KA Scripts are applicable (including often several instantiations of a same KA Script) is a task that can be done automatically. At any given time, there can be many errors in the knowledge base and several KA Scripts may apply for each error. ETM guides the user by suggesting KA Scripts that will resolve errors that occur earlier during problem solving. When several KA Scripts are applicable for the same error, we leave the choice up to the user, since the appropriateness of a choice may depend on information that is not readily available to the tool (e.g., user's preferred strategy to modify knowledge bases).

Preliminary Evaluations

We conducted some preliminary evaluations of our work by comparing the performance of several subjects using EXPECT and ETM with two different scenarios that required modifying a knowledge base. Both scenarios used the same knowledge base (from a simplified transportation domain), one of them (PAE) was slightly more complex than the other one (RTT). The scenarios and tools were used by the subjects in different order so that the results were not influenced by tiredness or increased familiarity with the domain. All of our users were familiar with EXPECT (but not with ETM), and had some previous exposure to the transportation domain. The subjects were first given some introductory material about the tools, the domain, and the kind of task to be done and were given a chance to practice using both tools. The experiment took several hours for each of the subjects, and we took detailed transcripts of what they were doing during that time. We also instrumented the tools to record the user's interactions, the errors in the knowledge base, and the time between each modification. The table below shows some results of these evaluations.

	RTT scenario				PAE scenario			
	EXPECT		ETM		EXPECT		ETM	
	S4	S1	S2	S3	S2	S3	S1	S4
Total time (min)	25	22	19	15	74	53	40	41
Time completing transaction	**16**	**11**	**9**	**9**	**53**	**32**	**17**	**20**
Total changes	3	3	3	3	7	8	10	9
Changes made automatically	n/a	n/a	2	2	n/a	n/a	7	8

The total time includes the time to understand the instructions for modification (which is is comparable for all subjects in the same scenario), and the time between the first change to the knowledge base and the completion of the transaction (i.e., to leave the knowledge base in a coherent state and succesfully computing a given set of sample problems). Subjects using ETM took consistently less time, the contrast is greater for the time to complete the transaction and in the more complex scenario (PAE). Notice that the subjects were familiar with EXPECT but not with ETM, which may be a factor in why some of them completed the modifications using EXPECT in times comparable to ETM in the simpler scenario. We expect the difference to be much larger in our future tests with users who are not familiar with EXPECT. The table also shows the number of changes done automatically by the KA Scripts, which may be one of the reasons why the subjects took less time with ETM.

It is interesting to note that in the longer scenario (PAE) both subjects using EXPECT had forgotten to perform part of the modification specified in the instructions. To realize that that was the cause for the wrong results that they got during the execution of the sampe problems, and to revert that situation (which sometimes requiered to redo part of the modification in a different way) took them considerable time. One possible explanation of why subjects using ETM did not have that problem is that ETM gives step by step guidance for modifying problem-solving methods, and relives users from keeping track of the pending changes, permiting the users to concentrate in the problem-solving method being modified. In contrast with our experience with previous versions, users were able to understand what the KA Scripts suggested and to follow the guidance that they provided in completing modifications. Although ETM allows users to abandon the KA Scripts and use EXPECT, none of our subjects decided to pursue this option.

Conclusions

We have described an approach to supporting users in modifying knowledge bases. The approach is based on identifying typical sequences of changes to a knowledge base and representing strategies (scripts) for carrying them out. These scripts allow the knowledge acquisition tool to understand the consequences of each individual change made by the user and provide support in completing the overall modification so that the knowledge base is not left in an unusable state.

One important extension to our approach is to incorporate scripts to help the user in starting modifications, not just completing them. In fact, three of our four subjects made the comment that they would like help in figuring out where to start the modification. These initiation scripts are similar to the programming cliches in the KBEmacs program editor (Waters 1985), which represent generic algorithmic fragments that programmers use in writing code.

Our initial implementation and preliminary evalua-

tions with users show promising results. We expect the benefits of KA Scripts to be greater for domain experts with no previous exposure to EXPECT or the domain implementation. Our user interface needs to be extended to provide visualizations and abstractions of the knowledge base as well as on-line help.

There are several features that make the EXPECT architecture suitable for supporting KA Scripts. EXPECT has explicit representations of all the knowledge in a knowledge-based system. These representations can be examined by ETM to understand which pieces of knowledge need to be changed. Other frameworks lack this kind of explicit representation, either because they use first-order logic representations that blur important distinctions among different types of knowledge, or because they hard-code some parts of the knowledge-based system reasoning, such as problem-solving knowledge (Eriksson *et al.* 1995). Another advantage of EXPECT is that it can analyze how generic goals (representative of the types of tasks that the knowledge-based system is built for) are achieved. Other frameworks lack this capacity, fording them to examine execution traces of specific problems that the system was unable to resolve, where relevant information for debugging the knowledge base is confounded in the details about that particular execution. Finally, EXPECT is built to handle errors in the knowledge base. When it encounters an error during problem solving, it generates a detailed description of how the error came about. It also has strategies for recovering from the error by using other information from the current knowledge base. Most systems are not built to handle faulty knowledge bases, often reporting errors that are both hard to understand and hard to fix. We believe that these architectural features are not only necessary to support KA Scripts, but also useful to address adequately the maintainance of knowledge-based systems.

Acknowledgements

We would like to thank Kevin Knight, Eric Melz, and Andre Valente for their valuable comments on this paper. Our special thanks to the past and present members of the EXPECT research group that patiently participated in our experiments, making possible the evaluation of ETM reported here. We gratefully acknowledge the support of DARPA with the contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative.

References

Eriksson, H.; Shahar, Y.; Tu, S. W.; Puerta, A. R.; and Musen, M. A. 1995. Task modeling with reusable problem-solving methods. *Artificial Intelligence* 79(1995):293–326.

Gil, Y., and Melz, E. 1996. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence.*

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

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

Langley, P., and Simon, H. A. 1995. Applications of machine learning and rule induction. *Communications of the ACM* 38(11).

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 Mateo, 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.

Murray, K. S. 1996. KI: A tool for knowledge integration. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence.*

Ourston, D., and Mooney, R. J. 1994. Theory refinement combining analytical and empirical methods. *Artificial Intelligence* 66:311–344.

Pazzani, M. J., and Brunk, C. A. 1991. Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning. *Knowledge acquisition* 3(2):157–173.

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.

Runkel, J. T., and Birmingham, W. P. 1993. Knowledge acquisition in the small: Building knowledge-acquisition tools from pieces. *Knowledge acquisition* 5(2):221–243.

Swartout, B., and Gil, Y. 1995. EXPECT: Explicit Representations for Flexible Acquisition. In *Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop.*

Waters, R. 1985. The programmer's apprentice: A session with kbemacs. *IEEE Transactions on Software Engineering* 11(11):1296–1320.