To appear in the International Journal of Human-Computer Studies

A Script-Based Approach to Modifying Knowledge-Based Systems

Marcelo Tallis

Department of Computer Science and Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292, USA, tallis@isi.edu

Abstract

Modifying knowledge-based systems is a complex activity. One of its difficulties is that several related portions of the system might have to be changed in order to maintain the coherence of the system. However, it is difficult for users to figure out what has to be changed and how. This paper presents a novel approach for building knowledge acquisition tools that overcomes some of the limitations of current approaches. In this approach, knowledge of prototypical procedures for modifying knowledge-based systems is used to guide users in changing all related portions of a system. These procedures, which we call *knowledge acquisition scripts* (or *KA Scripts*), capture how related portions of a knowledge-based system can be changed coordinately. By using KA scripts, a knowledge acquisition tool would be able to relate individual changes in different parts of a system, enabling the analysis of each individual change from the perspective of the overall modification. The paper also describes the implementation of ETM: a script-based tool that builds on the EXPECT framework for building knowledge-based systems (Gil, 1994), discusses how we have addressed some important issues of this approach, and describes a preliminary empirical evaluation of ETM that shows that KA Scripts allow users to perform knowledge-based systems modification tasks more efficiently.

1 Introduction

Modifying knowledge-based systems s) is a complex and recurrent activity. Events like changes in the system's environment, new user requirements, debugging of inadequate knowledge, and the customization of knowledge-based systems to user preferences and institutional practices are some of the causes that make knowledge-based systems modifications the rule rather than the exception. Unfortunately, modifying a knowledge-based system is a complex task. One of its difficulties is that after changing portions of the system, other related portions might also need to be changed to conform to the former changes. Determining what other portions have to be changed and how to change them to conform to previous changes requires a deep understanding of how the elements of the system interact. This requirement is especially hard to fulfill if the rationale behind the design of a system or the details of its implementation are unknown, which is often the case due to staff migration or third party software development.

One way of lessening the burden involved in modifying knowledge-based systems is by using *knowledge acquisition tools*. Most standard knowledge acquisition tools assume that the knowledge-based system follows a predefined inference structure or problem-solving method (PSM), and they only support the addition and modification of the domain-dependent knowledge required by the problem-solving method (Marcus & McDermott, 1989; Puerta, Egar, Tu, & Musen, 1992; Runkel & Birmingham, 1993). For instance, SALT (Marcus & McDermott, 1989) is a knowledge acquisition tool tailored to the propose-and-revise problem-solving method for solving system-configuration problems. SALT only supports the acquisition of the domain-dependent knowledge used by propose-and-revise such as procedures that propose initial values to different configuration parameters. By using knowledge that is specific to the problem-solving methods, these knowledge acquisition tools can provide a very strong support for modifying knowledge-based systems. However, its applicability is severely limited since a) they can only be used for a particular PSM, and b) they can only support modifications to the domain-dependent knowledge specified by the corresponding PSM.

There is another class of knowledge acquisition tools which provide support for a wider range of knowledge-based systems and modifications (e.g., (Gil, 1994; Yost, 1993)). These tools are independent of the method and do not restrict the type of modifications that can be done. The trade-off for this gain in flexibility is that they provide a weaker support. In EXPECT (Gil, 1994; Swartout & Gil, 1995), the expectations for guiding knowledge-based system modification are based on the understanding of the interactions among system components. Based in this understanding, EXPECT is able to detect inconsistencies in the knowledge-based system and to suggest changes that might fix them. However, these suggestions are generated regardless of what have caused the inconsistencies or what other changes have been done in other parts of the system. Consequently, these suggestions are too general to provide a precise guidance and the resulting support is weaker.

To develop a knowledge acquisition tool that both provides strong guidance to users and allow a wide range of modifications for any type of knowledge-based system, we investigated the use of an alternative source of expectations which is also problem-solving method independent: the understanding of prototypical procedures for modifying knowledge-based systems. These procedures, which we call *knowledge acquisition scripts* (or *KA Scripts*), capture how related portions of a knowledge-based system can be changed coordinately. They provide a context for relating individual changes to different parts of a knowledge-based system and hence enable the analysis of each change from the perspective of the overall modification.

We have developed an approach for supporting modifications of knowledge-based systems that uses a library of KA Scripts to guide a user in changing all related portions of the knowledge-based system in a consistent way. This approach is being implemented in a knowledge acquisition tool called ETM, a knowledge acquisition tool for modifying knowledge-based systems that builds on the EXPECT framework. ETM has been subject to some preliminary evaluations that have shown promising results.

In a previous paper (Gil & Tallis, 1995) we introduced the concept of a script-based knowledge acquisition tool and demonstrated it with an initial, simplified prototype. This paper presents recent findings from our work in designing and implementing ETM, a script-based knowledge acquisition tool. It discusses the research issues that we face and the right and wrong design decisions that we made. Additionally, it presents the results of our preliminary evaluation of ETM. This paper adds to (Gil & Tallis, 1997) the discussion regarding research issues and lessons learned from our experience in building ETM. It also contains a more comprehensive description of ETM.

The remainder of this paper is organized as follows. Section 2 discusses the difficulties involved in modifying knowledge-based systems and introduces an example that will be used throughout the paper. Section 3 introduces our script-based approach for supporting knowledge-based system modifications. Section 4 discusses the lessons learned through our experience in building a script-based knowledge acquisition tool. Section 5 presents some background on EXPECT, a framework for developing knowledge-based systems that we used for our work. Section 6 describes ETM, our implementation of a KA Scripts based tool. Section 7 elaborates the details of the knowledge-based system modification of section 2 and uses it to illustrate an application of ETM and its advantages over EXPECT. Section 8 describes the results of a preliminary evaluation that we conducted with several users. Finally, sections 9 and 10 discuss related work and conclusions.

2 Difficulties of knowledge-based system Modifications

Consider the following example from our experience concerning a knowledge-based system for transportation planning. Suppose that this system calculates durations of trips involving only ships, and that now it has to be extended to consider aircraft too. This modification requires several individual changes. First, existing knowledge may need to be modified. The description of vehicles used for trips has to be extended to include aircraft in addition to ships. The methods (or procedures) to estimate the duration of trips need to be changed to take into account aircraft. New knowledge may also need to be added. For example, new methods 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 method for the round trip time of aircraft and in the already existing calculation for the round trip time of ships, so we need to make sure that they use consistent procedures for estimating this distance.

Notice that if the user makes only some of these changes, the knowledge base system will be left in an *incoherent* state that will preclude its ability to calculate duration of trips. For example, suppose that the description of vehicles is extended to include aircraft, but no methods 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 will not be able to compute the round trip time of the aircraft. There are several reasons why it is hard for a user to complete a modification:

- **Changes in one element of the knowledge-based system affect to other elements:** Due to interdependencies among knowledge-based system elements, modifications to one element may require that other elements be modified, too. For example, the method for estimating the duration of a trip depends on the description of the vehicles used. When this description is modified, this method has to be modified, too.
- **Hidden interdependencies:** Interdependencies among knowledge-based system elements may not be explicit in the system. They are often dynamically determined by a problem solver (e.g., through a search process) and supported by system-made inferences (e.g., class inheritance). In our example, the method for calculating the round trip time of a ship was used for achieving an intermediate goal concerning duration of trips. A user unaware of this fact will not be able to recognize that there is an interdependency between this method and the description of the vehicles used in trips.
- **Cascading of changes:** After changing a knowledge-based system element other elements might need to be changed too. Furthermore, each of these additional changes can in turn originate the need for yet other additional changes. It is hard for a user to track down and to keep in mind all the changes that are pending.
- **Discrepancies in knowledge:** Changes and additions of knowledge should be made compatible with existing knowledge (i.e., should avoid redundancies or contradictions). One of the changes of our example is to include a new method for calculating the round trip time of aircraft. However, the existence of a method for calculating the round trip time of ships might suggest that generalizing this method (to cover aircraft) is a better solution than creating a slightly different and almost redundant method specifically for aircraft.

In sum, modifying a knowledge-based system often requires several individual changes to various components, and all those changes must be carefully coordinated. A good starting point is to build knowledge acquisition tools that find problems with the system 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, knowledge acquisition 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.

3 Script Based Knowledge Acquisition

Our approach is to equip a knowledge acquisition tool with *knowledge acquisition scripts* (or *KA Scripts*) that represent prototypical procedures for modifying knowledge-based systems. More specifically, a KA Script describes a prototypical sequence of changes for following up the consequences of previous changes to the knowledge-based system. An example of a KA Script is:

> if a) a posted goal is made more general (e.g., the goal *calculate the round trip time of a* **ship** is changed to *calculate the round trip time of a* **vehicle**) and
> b) the modified goal cannot be achieved by any existing method in the KB,
> then generalize the method that achieved the original goal.

This script might seem obvious. However, a knowledge acquisition tool lacking this knowledge could not discern whether to fix the problem described in part b) of the KA Script by generalizing a method (as the KA Script does) or by specializing the unachieved goal. The later would be incorrect because it would retract a previous change. Furthermore, this KA Script indicates which method has to be modified.

Because there is not a unique way of following up the consequences of a change, there can be several KA Scripts that apply to the same situation. For example, an alternative KA Script that applies in the previous example is to create a new method for achieving the generalized goal rather than to change an existing method. KA Scripts can also be specific to particular situations. For example, a KA Script that applies in the previous examples, but in a more particular situation is[1]:

> if a) a posted goal is made more general (e.g., the goal *calculate the round trip time of a* **ship** is changed to *calculate the round trip time of a* **vehicle**) and
> b) the modified goal cannot be achieved by any existing method, and
> c) the generalized goal now can be decomposed into two disjunct subcases (e.g, the goal*calculate the round trip time of a* **vehicle** can be decomposed into one case for **ship** and another case for **aircraft**), and
> d) one of these subcases is equivalent to the goal before being generalized
> then 1) duplicate the method that achieved the goal before being changed (this method can still achieve one of the subcases), and
> 2) adapt this copy to the other subcase (e.g., duplicate the method for **ships** and adapt it to **aircraft**). This way, the two methods combined would achieve the modified goal.

[1]A more detailed description of this KA Script is presented in Section 6 and shown in Figure 3.

A script-based knowledge acquisition tool suggests and executes KA Scripts that take care of the consequences of previous changes. To support this functionality a knowledge acquisition tool needs to have access to the following kinds of information:

- **Problems within the current knowledge-based system** (e.g., inconsistencies) which might be indicative of unattended consequences of previous changes. Examples of such problems are goals that cannot be achieved and references to non-existing attributes of domain concepts. These problems do not mean that previous changes are a mistake but rather that the ongoing modification is still incomplete. The knowledge acquisition tool uses this information to suggest the execution of KA Scripts that will take care of these consequences. In our knowledge-based system modification example, the lack of a method for calculating the round trip time of aircraft is a consequence of the change in the definition of vehicles. Noticing this, the knowledge acquisition tool could suggest, among other options, a KA Script that creates a method for calculating the round trip time of aircraft based on the one used for ships.
- **A history of the changes made to the knowledge-based system** which the tool can use to infer what the user is trying to accomplish with the modification and to avoid considering KA Scripts that interfere with those intentions. Continuing with our example, the knowledge acquisition tool could observe that the definition of vehicles has been recently changed and hence avoid suggesting any KA Script that attempts to change this definition back to its original form.
- **A record of past versions of the knowledge-based system** which the tool can use to understand how knowledge-based system elements used to fit together. For example, observing which method had been used to calculate the round trip time of ships allows the tool to consider a KA Script that manipulates this same method in order to achieve the round trip time of aircraft.

In designing a script-based knowledge acquisition tool, the following research issues were found to be important.

- KA Scripts should be at an appropriate level of generality. Overly general KA Scripts do not provide adequate guidance for users to follow. For example, a KA Script that indicates "Modify a method in order to compute the round trip time of a vehicle" will not be as useful as a KA Script that indicates "generalize the method for computing the round trip time of a *ship* to make it applicable for *vehicles*." In addition, the sequence of steps in a KA Script should be ordered to maximize the support that the tool could provide in executing these steps. Sometimes, the way a user makes a change sheds some light on how she may go about making other changes. Consequently it is preferable to place earlier any changes that can be analyzed to guide subsequent changes.
- The tool needs a mechanism to prioritize and arbitrate among all the KA Scripts that apply to the current situation. Several KA Scripts could be applicable not only because a given problem may accept different solutions but also because several problems could be present at the same time. A script-based knowledge acquisition tool needs to determine which problem to attend first since all problems might be related and the order in which they are handled may simplify or complicate the task.
- Users should have the flexibility to modify a knowledge-based system either by using KA Scripts or by performing individual changes. Sometimes the user already knows the changes

that she wants to perform and there is no point in forcing her to find a KA Script that would perform them. Additionally, it is unlikely that a KA Scripts library contains all possible strategies for fixing problems and the tool should let the user follow strategies that are not in the library.

- In completing a modification, the tool should guide a user throughout a logical sequence of changes that has to be easy to follow and understand. The tool should not interrupt the flow of the changes that the user is performing because that would cause the user to lose the thread of what she was doing.

We have learned about these issues through our experience in building a script-based knowledge acquisition tool. Our earlier attempts for building a script-based knowledge acquisition tool showed some deficiencies that were finally corrected in our current implementation. Next section summarizes some of the design decisions that led us to the construction of a successful script-based knowledge acquisition tool, and some that did not.

4 Designing a Script-Based Knowledge Acquisition Tool

To elaborate our library of KA Scripts, we first did a thorough analysis of the kinds of changes that can be done to a knowledge-based system, their possible consequences, and the successive changes that could follow up on those consequences. This analysis was done systematically by evaluating the consequences of modifying every constituent in the constructs used to represent knowledge in our framework. For example, we analyzed different types of changes to goals (e.g., modify one parameter), their possible consequences (it might not be possible to match that goal to a method), and their follow-up changes (e.g., change that same parameter in the method that achieved that goal before, and then change the body of that method accordingly). The result of this analysis was a complete set of KA scripts that followed up all possible consequences of changes. These KA Scripts cover all the situations in which a user can get when modifying a knowledge base. However, tests with our initial implementation showed that the guidance they provide to users is very general. The main problem is that they do not make good use of the context available, like particularities of the knowledge base (e.g., that the modified goal can be decomposed into two subcases) or the changes performed to other parts of the knowledge base.

To overcome this problem we tried a different approach. We analyzed several sample scenarios for modifying knowledge based systems. We looked at the changes that needed to be made, their consequences, and how subsequent changes followed those consequences. 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. We plan to continue finding more KA Scripts using this method.

Finally, KA Scripts produced by both methods were combined into a single library. Hence, there is always at least one general KA Script in the library that applies to a given situation. In situations where a more specific KA Script can also be applied, the guidance provided will be more specific to the context and more precise. The result of this effort was a library of 75 KA Scripts from which only a dozen have been implemented so far.

A KA Script library needs to have some structure to guide its development and to allow the tool to access relevant KA Scripts without examining the whole library. In our current implementation,

Problem Types	
unmatched Goal	empty WHEN block
no desired matching of goal	non boolean condition in WHEN form
unused result of expression	non SET argument in FILTER form
invalid argument of relation	non boolean condition in FILTER form
less than two operands in logical form	less that two args in APPEND form
unnecessary grouping in logical form	incompatible type of args in APPEND
empty THEN block	unused variable
empty ELSE block	undeclared variable
empty THEN and ELSE block	method too complex
non boolean condition in IF form	disagreement in result type of method
incompatible THEN and ELSE types	unused method

TABLE 1: Type of problems addressed by a KA Script library used to guide modifications of EXPECT knowledge bases

KA Scripts are discriminated by the type of problems (or errors) that a KA Script could fix (e.g., a goal cannot be matched) and can further be discriminated by their requirements over past changes (e.g., a goal became more general) or the content of the KB (e.g., the unmatched goal can now be decomposed). Table 1 list the type of problems addressed by our KA Script library.

Our first attempt was to structure the library around type of changes to be followed up, but this produced scripts that were very cumbersome. The problem is that these changes can have very different consequences depending on the situation. Moreover, it turns out that the procedures for following up those changes are more dependent on these consequences that on the type of the change itself.

We also realized that the sequence of steps of a KA Script has to be short and simple, and not complicated with provisions for handling all possible consequences of its changes. It is better to attend these consequences in separate KA Scripts. The rationale behind this decision was also that a change can have very different consequences depending on the situation, and it is impractical to include provisions for handling all of them in every KA Script. In addition, shorter KA Scripts are easier for a user to understand and follow.

We let the execution of a KA Script to reach its end before attending the consequences of changes performed during its execution. Otherwise it would produce a high degree of nesting that would make it hard for users to follow what is happening. Another reason is that some of the problems generated in the middle of the execution of a KA Script are fixed naturally when the KA Script finishes its execution.

To avoid that the user lose the thread of what she is doing and to give her some control over the process, the user takes the initiative to start the execution of a KA Script when she considers appropriate (although the tool suggests which KA Scripts can be applied). In addition, she is free to make any change to the knowledge-based system between the execution of KA Scripts without being interrupted by the tool. To help the user follow the thread of the KA script without getting lost, the user interface shows all the steps in the current KA Script with indications of which steps have already been executed, which one is being executed now, and which are still pending.

We have built a script-based knowledge acquisition tool called ETM. ETM is a tool for supporting modifications of EXPECT knowledge-based systems which was also built integrated to the EXPECT system (Swartout & Gil, 1995; Gil, 1994).

5 EXPECT: a knowledge-based system Framework and Baseline knowledge acquisition tool

EXPECT is an environment for building and modifying knowledge-based systems. EXPECT provides three key capabilities: a knowledge-based system representation framework, a problem solving environment, and a knowledge acquisition tool. We introduce some aspects of EXPECT as we present an example knowledge base that is going to be used throughout the rest of the paper. More details about EXPECT can be found in (Gil & Melz, 1996; Swartout & Gil, 1995; Gil, 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 from the KL-ONE family. Problem solving knowledge is represented in *methods* which are procedural descriptions for achieving goals. They consist of 1) a *capability* that represents the general 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 1 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 (a subgoal that has to be achieved by other methods) and divide it by the speed of the ship.

Given a generic goal (referred to as the *top-level goal*) such as

$$(\textit{estimate} \quad \text{(obj (spec-of TRIP-DURATION))} \\ \text{(of (inst-of TRANSPORTATION-MOVEMENT)))} \tag{1}$$

estimate the trip duration of a transportation movement

EXPECT's problem solver automatically generates a domain specific knowledge-based system for solving instances of this goal, such us:

$$(\textit{estimate} \quad \text{(obj (spec-of TRIP-DURATION))} \\ \text{(of FIRST-MOVEMENT-FT-BRAGG-TO-RYAD))}$$

estimate the trip duration of the first movement from Fort Bragg to Ryad

Problem solving proceeds as follows: the top-level goal is expanded 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 constraints on the goal parameters performing an elaborate form of partial evaluation which is supported by Loom's reasoning capabilities. The derivation tree is then used by the EXPECT knowledge acquisition tool to support knowledge acquisition. One strong capability of EXPECT is that problem-solving is performed with generic goals (such as (1)). Hence, if EXPECT is able to solve the generic problem then it would be able to solve any instance of it. In addition, EXPECT can identify what knowledge is needed for solving problem instances. This ability drives knowledge acquisition.

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

A vehicle is a kind of major equipment and has a speed. A vehicle is either a ship or an aircraft.

```
(defconcept TRANSPORTATION-MOVEMENT
  :is-primitive
       (:and (:the HAS-ORIGIN LOCATION)
             (:the HAS-DESTINATION LOCATION)
             (:the HAS-CARGO-WEIGHT-TO-MOVE TONS)
             (:some HAS-AVAILABLE-LIFT SHIP)))
```

A transportation movement has an origin and a destination location, a cargo to move expressed in tons, and some lift available consisting in ships.

```
(def-expect-method M1
 (capability
   (estimate
     (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  (spec-of ROUND-TRIP-TIME))
                (by   (HAS-AVAILABLE-LIFT ?m))
                (from (HAS-ORIGIN ?m))
                (to   (HAS-DESTINATION ?m)))))))
```

To estimate the trip duration of a transportation movement first calculate the round trip time, from the origin to the destination of the movement, for each one of the lift available for that movement, then pick the longest.

```
(def-expect-method M2
 (capability
   (calculate (obj  (?t is (spec-of ROUND-TRIP-TIME)))
              (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)))))
```

To calculate the round trip time between two locations for a ship first find the sailing distance between the two locations, then divide this distance by the speed of the ship.

```
(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)))))
```

To find the sailing distance between two locations, if any of the locations is unknown then ask the user for that distance, else look up the table of sailing distances.

FIGURE 1: Some definitions of concepts and problem solving methods in a simplified transportation domain.

In our example, method M1 needs to calculate the round trip time for the lift available for a transportation movement (subgoal CALCULATE). EXPECT will look at the role HAS-AVAILABLE-LIFT in definition of TRANSPORTATION-MOVEMENT and determine that the lift of a movement includes only ships, hence this goal can be achieved using method M2. In turn, method M2 needs to find the sailing distance between two locations (subgoal FIND), which can be achieved using method M3.

Using the derivation tree, EXPECT finds the interdependencies between domain facts and problem-solving methods, which are used to guide knowledge acquisition. For example, 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 be considered as an error and the knowledge acquisition tool will ask the user to specify the speed. These interdependencies also help EXPECT's knowledge acquisition tool to detect errors or potential problems in the KB, such as 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.

All KB errors found by EXPECT are recorded in an *agenda* that reminds the user of what has yet to be fixed. EXPECT supports users in resolving these agenda items by explaining these errors and suggesting fixes to them. Besides the handling of errors, the EXPECT knowledge acquisition tool can also display different graphical presentations as well as natural-language descriptions of the KB and provides a menu interface for executing commands that modify the KB.

```
User makes change(s) in the knowledge base
EXPECT posts in its Agenda the problems found in the KB
While there are items left in the Agenda
      ETM picks problem i from the Agenda (the one that occurs
            first) and generates set K of KA Script
            candidates that can fix i
      If the user does not choose any KA Script
        then user can quit ETM and fix i with EXPECT,
             ETM can be invoked again anytime
        else user chooses k from the set K,
             ETM and user apply k
      EXPECT updates its Agenda
```

FIGURE 2: ETM's Control Loop

6 ETM: a Script Based knowledge acquisition Tool for EXPECT

We have implemented a Script-based knowledge acquisition tool that supports modifications of EXPECT knowledge-based systems. This tool, called *Expect Transaction Manager*[2] (or *ETM*), was built tightly integrated to the EXPECT's architecture for two reasons: a) to make use of EXPECT built-in analytical capabilities, in particular determination of interdependencies and identification of errors, and b) to allow users to use both, EXPECT's conventional tool and the script based tool. This last feature is essential because a user may always come up with new strategies to modify the KB that are not contained in the KA Script library. To perform this integration, EXPECT required only slight extensions. Namely, EXPECT's knowledge acquisition tool was modified to keep record of past versions of the KB and the history of modification commands executed so far.

We built ETM to help a user to bring a knowledge-based system back to a coherent state after a user had performed some initial changes to the system using EXPECT conventional knowledge acquisition tool (the system is assumed to be coherent before starting its modification). ETM engages in a loop in which 1) it identifies unattended consequences of previous changes, 2) suggests KA Scripts that take care of those consequences, and 3) guides the user throughout the application of one of these KA Scripts (chosen by the user). The completion of a modification usually requires of the execution of several KA Scripts. This is because some changes can have several consequences that have to be followed up by different KA Scripts, and also because of the cascading of changes (i.e., changes that followed up consequences of previous changes have their own consequences that require being followed up).

The overall control loop for the execution of KA Scripts is shown in Figure 2. The loop begins after the user has executed some changes to the knowledge-based system and requested the help of ETM to follow up the consequences of those changes. For each problem in the knowledge-based system caused by previous changes, ETM proposes KA Scripts that apply to that situation. If the user agrees with one of them then ETM, with the help of the user, executes it. If not, the user fixes the problem using EXPECT's basic knowledge acquisition tool.

Our goal is to automate as much as possible the application of KA Scripts. Although full automation

[2]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

```
Problem to be fixed: "Goal G-new cannot be matched"

Applicability Conditions:
   (a)  Past change has caused argument A of 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 subgoals {G1 G2}
   (d) G1 is equivalent to G

Modification sequence:
   CHOICE 1: Create new method M-new based on existing method
     (1) Choose a method to be used as basis.
           System proposes M. User may accept it or choose another.
     (2) Create method M2-new analogous to the basis.
           System creates M2-new as a copy of the basis and proposes
           substituting A to match G2. User may generalize further and
           indicate additional substitutions needed in the body of
           M2-new.
     (3) Edit body of M2-new if modifications other than substitutions
         are needed.
           User edits body of M2-new.
   CHOICE 2: Create new method M2-new from scratch

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

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

FIGURE 3: KA Script example

is not possible, the tool can take care of significant parts of the process, leaving important high-level decisions to the user. Detecting which KA Scripts are applicable (including often several instantiations of a same KA Script) is a task that can be done by the tool. At any given time, there can be many problems in the knowledge base and several KA Scripts may apply for each problem. ETM suggests to start by problems that occur earlier during problem solving because their fixes will determine how the problem-solving process will continue. When several KA Scripts are applicable for the same problem, 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). Finally, the execution of a KA Script is shared between the knowledge acquisition tool and the user.

Figure 3 shows in detail one of the KA Scripts in our library[3]. The representation of KA Scripts includes the following kinds of information:

- **Problem to be fixed:** The type of problem or error in the KB that the KA Script is intended to fix (see table 1). This information is used to organize and index KA Scripts into a library.
- **Applicability conditions:** Conditions that specify appropriate situations for applying the KA Script. These conditions are based on the content of the current knowledge-based system as well as previous states of the knowledge-based system and history of changes. There are

[3]Note that KA Scripts are implemented in Lisp. Figure 3 is a natural-language description of a KA Script procedure.

two types of conditions: A) preconditions required to achieve the KA Script purpose (e.g., conditions (b), (c) and (d) in Figure 3), and B) conditions for ensuring that the KA Script is a sound continuation for the ongoing modification (e.g., condition (a)). The applicability conditions also bind variables to knowledge-based system elements, which result in an instantiated KA Script.

- **Modification Sequence:** The sequence of changes to be performed. Sometimes it is not possible to indicate specific steps that can be executed automatically and have to be partially specified (e.g., steps 2 and 3). In those cases, the user is responsible for refining these steps. However, ETM still supports the user by giving step by step instructions and suggestions based on the content of the knowledge-based system and the history of the modification. Toward this end, the sequence of steps in KA Scripts is designed so that the execution of the preceding steps provides useful information to help the user to execute the successive steps. The order is also concerned with making the sequence understandable by the user by grouping changes that are conceptually related.

- **Short Description of what the KA Script does:** A text template that describes what the KA Script does. This description is shown to the user to let her decide which KA Scripts she wants to use. The template will be filled with concrete references to elements of the knowledge-based system. This turns out to be very important because it is only after knowing these elements that the user can make such a decision.

- **Explanation of why the KA Script is relevant to the current situation:** An explanation that relates this KA Script to the ongoing modification. This explanation can be requested by the user if she wants to know why this KA Script is suggested. It is a text template that is filled with concrete references to the elements of the knowledge-based system.

7 An Example Scenario for KA Scripts

Suppose that the knowledge-based system of Figure 1 has to be modified because the available lift for transportation movements can now include aircraft besides ships. This modification requires five changes as depicted by Figure 4. First, in the definition of TRANSPORTATION-MOVEMENT, the range of the role HAS-AVAILABLE-LIFT has to be generalized to VEHICLE, which already includes both SHIP and AIRCRAFT. This is achieved with Action 1 (see Figure 4). The fillers of this role are one of the arguments of the goal CALCULATE in method M1, hence this argument changes automatically from SHIP to VEHICLE too. After this change, the goal CALCULATE in M1 cannot be matched any more with M2 (M2 applies only to ships). However, the new goal can be decomposed into two disjunctive subgoals, one for ships and the other for aircraft. In order to achieve the ship subgoal, the method that achieved it originally (M2) can still be used. This same method can also serve as a basis for constructing a new method for the aircraft subgoal. With Actions 2 and 3 the user creates this new method (M2-PRIME) by duplicating M2 and then modifying its copy (substituting SAILING-DISTANCE by FLYING-DISTANCE and adding the computation of the time that the aircraft spend in enroute stops). The new method contains two goals that cannot be achieved with the current knowledge: a modification of the goal FIND, whose argument SAILING-DISTANCE was changed to FLYING-DISTANCE, and the new goal COMPUTE. The user, with Action 4, creates a method (M3-PRIME) for achieving the modified FIND based on the method that achieved the original FIND (M3), and with Action 5 a new method (M4) for achieving COMPUTE.

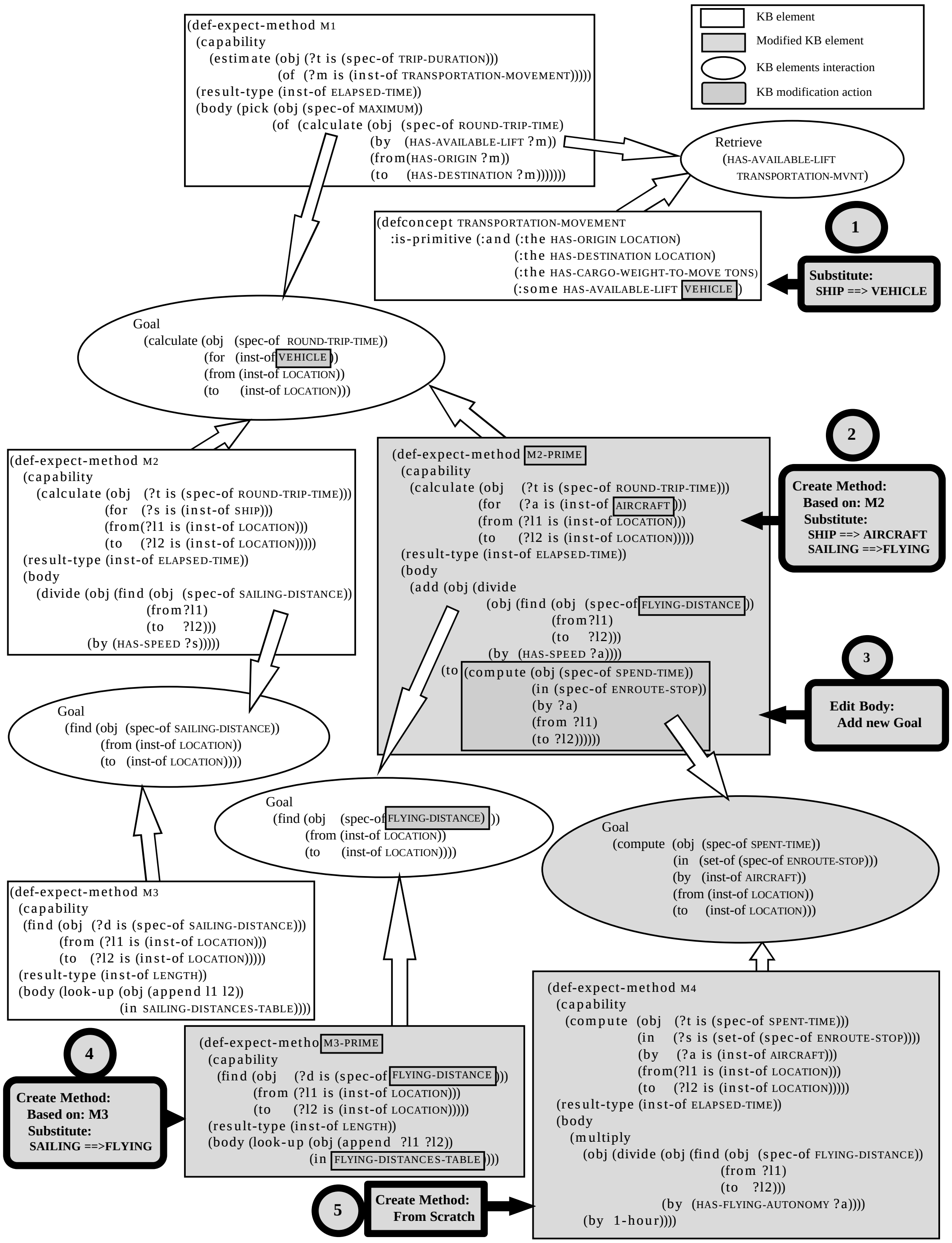

FIGURE 4: knowledge-based system modification scenario

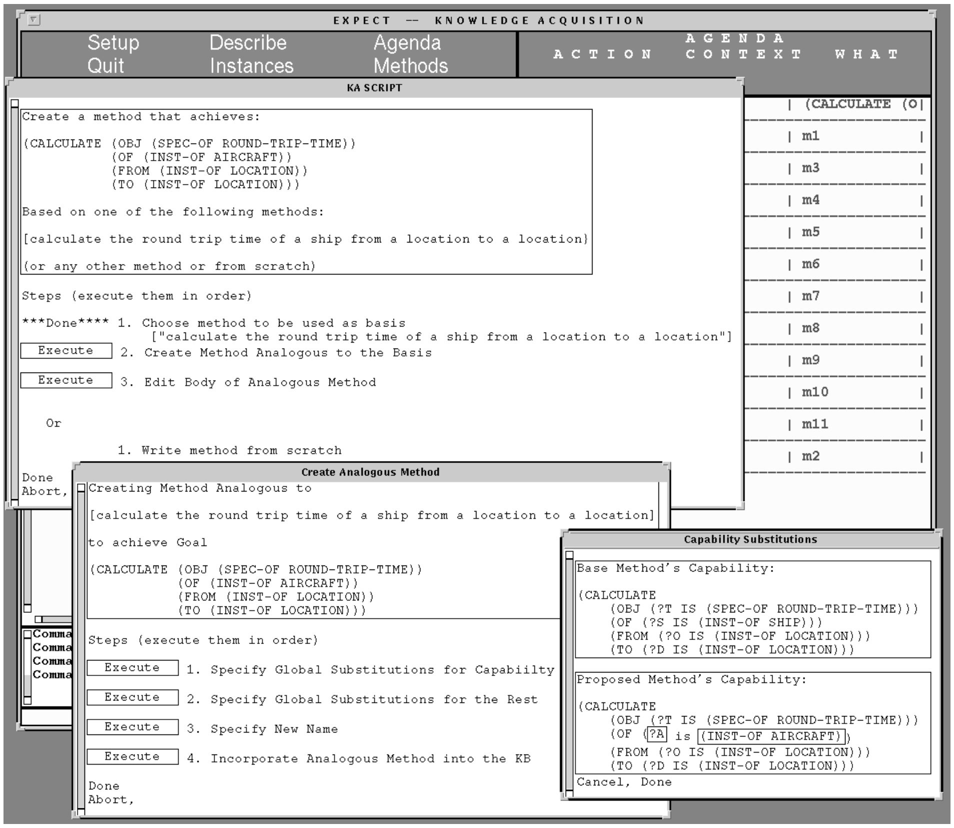

FIGURE 5: Snapshot of ETM's User Interface. The EXPECT's knowledge acquisition tool window is in the background overlapped by three other windows displayed by ETM during the execution of the KA Script of Figure 3. The back-most window shows the changes sequence of the KA Script. This snapshot was taken during the execution of change 2 (create a method analogous to the basis). On top of this window there is a window that guides the user step by step through the execution of this change. Finally, the front-most window is an ETM's suggestion for the first step (specify global substitutions for capability). Namely, substitute SHIP by AIRCRAFT and ?S by ?A.

In sum, five changes in different parts of the KB were needed to modify the knowledge-based system. If some of these changes were omitted, the system would have been left incoherent. It is important to realize that these changes are difficult for a user to determine. For example, to determine that after generalizing the range of the role HAS-AVAILABLE-LIFT (Action 1), a method for calculating the round trip time for an aircraft has to be created (Action 2), the user needs to determine the following facts: 1) The fillers of the role HAS-AVAILABLE-LIFT are used in one parameter of the goal CALCULATE of method M1, 2) CALCULATE was achieved by M2, 3) M2 cannot achieve the variation of CALCULATE, and 4) CALCULATE can be decomposed into a subgoal for ships and another for aircraft. Many of these facts are supported by logic inferences like goal subsumption (e.g., whether goal CALCULATE can be decomposed into disjunctive subcases or not) what makes it even harder for users to figure it out.

The KA Script of Figure 3 could be applied to fix the problem caused by the first change of this scenario, "Goal CALCULATE (of M1) cannot be matched", and would suggest to the user Actions 2 and 3 of that scenario. It is applicable because: Action 1 caused that argument SHIP of CALCULATE changed to VEHICLE (condition (a)), this goal was achieved by M2 before (condition (b)), and now it can be decomposed into two subgoals, one for SHIP and another for AIRCRAFT (conditions (c) and (d)). Figure 5 shows ETM's user interface during the application of this KA Script.

EXPECT's original support for accomplishing this modification is limited. EXPECT reports the errors that are present in the KB and suggests solutions to them. However, these suggestions are too general. EXPECT generates suggestions from a model that relates errors with possible fixes. However, error information alone is not enough to provide precise suggestions. An understanding of how this error was originated as well as of how its fix will contribute to the overall modification is also required. For example, after the first change of this scenario, EXPECT will report that the goal CALCULATE in M1 cannot be achieved and will suggest three possibilities: 1) modifying M1, 2) modifying some other method's capability, and 3) creating a new method. It cannot specify that it is method M2 the one that has to be modified in suggestion (2) because it does not know which method used to achieve goal CALCULATE before this change. It also cannot suggest the creation of a method for AIRCRAFT based on the method for SHIP because it is a fix that only applies to goals being made more general, and EXPECT does not know that that was the cause of the error.

8 Preliminary Results

We conducted some preliminary evaluations of our work by comparing the performance of four 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 the scenarios (PAE[4]) was slightly more complex than the other one (RTT[5]). Each subject had to solve both scenarios, one of them using EXPECT and the other using ETM. 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. For these preliminary evaluations, all of our subjects 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 were given a chance to practice using both tools. For each scenario, they were given instructions for the modification and a sample test problem with the expected solution after the modification. 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
Total changes	3	3	3	3	7	8	10	9
Changes made semi-automatically	n/a	n/a	2	2	n/a	n/a	7	8

The total time includes the time to understand the task and the time to complete the modification (i.e., to leave the knowledge base in a coherent state and successfully computing a given set of sample problems). According to our expectations, subjects using ETM took consistently less time and the contrast was greater in the more complex scenario (PAE). Notice that the subjects were familiar with EXPECT but not with ETM. 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 semi-automatically by the KA Scripts, which may be one of the reasons why the subjects took

[4]PAE stands for Personnel Aspect Evaluation. This scenario consisted in refine a rough estimation of the personnel needed for a trip, by discriminating personnel requirements depending on the type of seaports.

[5]RTT stands for Round Trip Time. This scenario consisted in adding aircraft to the definition of the vehicles available for a transportation (which originally consisted only in ships).

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 which was the problem that was causing the wrong results that they got during the execution of the sample problems, and to revert that situation (which sometimes required 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 KA Scripts present "checklists" of changes that the user should consider making.

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. Although ETM allows users to abandon the KA Scripts and use EXPECT, none of our subjects decided to pursue this option.

9 Related Work

There is a vast number of techniques for assisting the development of knowledge-based systems that, like ETM, follow the paradigm of detecting problems in a knowledge-based system and suggesting repairs. These techniques have been used in *knowledge base refinement* (e.g., TEIRESIAS (Davis, 1979), *theory revision* (e.g., EITHER (Ourston & Mooney, 1994), FOCL (Pazzani & Brunk, 1991)), *apprenticeship systems* (e.g., ODYSSEUS (Wilkins, 1990) and NeoDISCIPLE (Tecuci, 1992)), and *Validation & Verification* (e.g., (O'Keefe & O'Leary, 1993)). However, none of them uses a concept similar to our scripts or even considers the context in which errors were produced. This is because the purpose of those techniques is different than ours. The above mentioned approaches attempt to correct the errors in a knowledge-based system when it is being developed. In contrast, the purpose of ETM is to guide a user in modifying a knowledge-based system that is initially correct. In the case of ETM, the errors to be corrected were introduced during the modification process. Hence, ETM can rely on knowledge about this process in correcting these errors.

Some knowledge-based software engineering (KBSE) tools have incorporated a concept similar to our KA Scripts. KBEmacs (Waters, 1985) is a knowledge-based program editor that permits the construction of a program by combining algorithmic fragments (called *cliches*) from a library. KBEmacs cliches are equivalent to ETM KA Scripts except that cliches are algorithmic fragments that are used to *generate* programs while KA Scripts are procedures that are used to *modify* knowledge-based systems. Another difference is that KBEmacs does not support the selection and application of cliches while ETM automatically detects opportunities for using KA Scripts, suggests what KA Scripts can be applied, and guides users in applying a KA Script. An interesting point worth noticing is that both KBEmacs and ETM allow a user to perform modifications not supported by their procedure libraries by letting her to resort to a more general purpose tool (Emacs and EXPECT respectively) .

Another example of a script-based KBSE tool is the Knowledge-based Software Assistant (KBSA) and its successor ARIES (Johnson & Feather, 1991; Johnson, Father, & Harris, 1992). The purpose of these tools is to provide integrated support for requirements analysis and specifications development. They provide a library of *evolution transformations* that a user can apply to make consistency-preserving modifications to the description of a software system. These evolution transformations are similar in spirit to ETM's KA Scripts. Their main distinction lies in that evolution transformations are used to manipulate a semi-formal description of a system, while ETM modifies the actual implementation of a system. In the system's representations manipulated by KBSA and Aries, the interactions between the system elements are stated explicitly. In contrast, in the system's representations manipulated by ETM, the interactions between system elements are dynamically established by a problem solving process. This demanded of ETM the use of special techniques like the analysis of interdependencies among knowledge-based system elements or the analysis of the past states of a knowledge-based system to determine how system elements used to relate before. Our ultimate goal is to enable end users to modify a knowledge-based system by themselves. For this reason we have incorporated some techniques tailored to support end-users that KBSA and Aries do not have. First, we keep a history of previous changes to the system in order to interpret the user intentions and to avoid giving advice that may contradict them. Second, we included KA Scripts that are specific to particular situations. Although these KA Scripts lack generality, they provide a more precise and hence more helpful guidance.

10 Conclusions

Our goal is to develop an approach for building knowledge-acquisition tools that provide strong support for a wide range of modifications and knowledge-based system types. To achieve this goal, we equipped a knowledge-acquisition tool with a library of knowledge-acquisition scripts (or KA Scripts) which represent prototypical procedures for modifying knowledge-based systems. A knowledge-acquisition tool uses these KA Scripts to guide users in following up the consequences of changes performed to a . The advantage of KA Scripts is that they provide a context for relating individual changes to different parts of the knowledge-based system enabling the tool to analyze each change from the perspective of the overall modification. This kind of analysis complements previous approaches for interpreting changes to a knowledge-based system and enables a knowledge-acquisition tool to provide a more precise guidance. Because KA Scripts are problem-solving method independent, they can be used to support modifications of any kind of knowledge-based system. Furthermore, because KA Scripts represent varied procedures for modifying different aspects of a knowledge-based system, they can support a wide range modifications.

We have implemented a script-based knowledge-acquisition tool called ETM that supports modification to knowledge-based systems developed within the EXPECT framework. In implementing ETM we addressed several research issues that concerned the development of a KA Script library, the coordination of KA Scripts, and the model of interaction with the user.

To evaluate ETM, we carried out an experiment that compared the performance in modifying knowledge-based systems for subjects using ETM vs. subjects using EXPECT. The experiment showed that subjects using ETM outperformed the ones using EXPECT, especially in the more complex modification tasks. In this first experiment we chose subjects that were already familiar with EXPECT but not with ETM. We expect that the difference in performance will be more significant in our future experiments involving subjects not familiar with EXPECT.

One important extension to our approach is to give advise on how to start a modification, not just how to complete it. In fact, three of our four subjects made the comment that they would like help in figuring out where to start the modification. One way to achieve this goal is by integrating KA Scripts of different level of abstraction. The more abstract KA Scripts would plan the overall modification while the more specific ones would take care of the details.

Acknowledgments

I would like to thank Yolanda Gil, Ion Muslea, 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

Davis, R. (1979). Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12(2):121–157.

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

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

Gil, Y. & Tallis, M. (1995). Transaction-Based Knowledge Acquisition: Complex Modifications Made Easier. In *Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada.

Gil, Y. & Tallis, M. (1997). A script-based approach to modifying knowledge-based systems. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, RI.

Johnson, W. L., Father, M. S., & Harris, D. R. (1992). Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869.

Johnson, W. L. & Feather, M. S. (1991). Using evolution transformations to construct specifications. In *Automating Software Design*, pp. 65–92. AAAI Press.

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

O'Keefe, R. M. & O'Leary, D. E. (1993). Expert System Verification and Validation: A Survey and Tutorial. *Artificial Intelligence Review*, 7:3–42.

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

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

Swartout, B. & Gil, Y. (1995). EXPECT: Explicit Representations for Flexible Acquisition. In *Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada.

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.

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

Wilkins, D. C. (1990). Knowledge base refinement as improving an incorrect and incomplete domain theory. In *Machine Learning: An Artificial Intelligence Approach*, volume 3, pp. 493–513. San Mateo, CA, Morgan Kaufmann.

Yost, G. R. (1993). Acquiring knowledge in soar. *IEEE Expert*, 8(3):26–34.