Designing Scripts to Guide Users in Modifying Knowledge-based Systems

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

Abstract

Knowledge Acquisition (KA) Scripts capture typical modification sequences that users follow when they modify knowledge bases. KA tools can use these Scripts to guide users in making these modifications, ensuring that they follow all the ramifications of the change until it is completed. This paper describes our approach to design, develop, and organize a library of KA Scripts. We report the results of three different analysis to develop this library, including a detailed study of actual modification scenarios in two knowledge bases. In addition to identifying a good number of KA Scripts, we found a set of useful attributes to describe and organize the KA Scripts. These attributes allow us to analyze the size of the library and generate new KA Scripts in a systematic way. We have implemented a portion of this library and conducted two different studies to evaluate it. The result of this evaluation showed a 15 to 52 percent time savings in modifying knowledge bases and that the library included relevant and useful KA Scripts to assist users in realistic settings.

Introduction

Developing Knowledge Acquisition (KA) tools that help users create and maintain knowledge bases (KBs) is an important research issue for AI. Once a prototype knowledge base is initially developed, users would like to maintain and extend it throughout its lifetime. Users need KA tools to guide them make these changes, because it is hard for them to foresee and follow up on all the effects and implications of each individual modification that they make. Script-based KA tools (Gil & Tallis 1997) help users follow typical modification procedures (*KA Scripts*), ensuring that they follow up the effects of each individual change and complete the overall modification. This kind of approach has also been shown to be useful for developing software (Waters 1985; Johnson & Feather 1991; Johnson, Feather, & Harris 1992), and for developing intelligent assistants for common tasks (e.g., the wizards that are now a part of many commercial tools).

Previous work has showed promising results with a scripts-based tool that contained a limited set of KA Scripts (Gil & Tallis 1997). We set out to scale up this approach and develop a library of KA Scripts that would provide a more extensive coverage of situations when users modify knowledge bases. We found that there are many ways to define these Scripts, that it is hard to find the appropriate level of generality, and that it is challenging to generate KA Scripts systematically to ensure good coverage. This paper reports our findings in all these areas. The paper also describes three different analyses that we conducted, each unveiling different insights about how to populate a KA Script library.

First, we carried out an analysis of the follow-up procedures for every possible syntactic modification to the elements of a knowledge-based systems (KBSs). The result was a set of KA scripts that has proven to be complete in its coverage but too general to provide useful guidance to users.

Second, we carried out a detailed analysis of KBS modification scenarios. The results obtained from this analysis were the reverse of those from the prior analysis, which was not satisfactory either. Using this method we obtained KA Scripts that proved to be very useful to users. Unfortunately, this method cannot produce a complete set of KA Scripts. There are few other reports in the literature of what kinds of modifications are made to knowledge bases, and they are often anecdotal. We report the results of this analysis in detail, so other researchers can benefit from this empirical perspective on what are typical knowledge acquisition tasks.

Finally, we combined both analyses to conceive a method that turned to be satisfactory in both, coverage as well as user support. The problem with our first analysis was that an indication of a syntactic change to a KBS element was too general to determine an operative procedure for following up that change. We needed a more specific description of the change as well as of the situation in which that change was performed and the strategy chosen to follow up that change. These more specific descriptions could be generated by combining all possible values for a set of defining attributes

of a KA Script. A set of attributes was determined by analyzing KBS modification scenarios.

We used this set of attributes to generate systematically a subset of the KA Scripts library, which we implemented. Our preliminary tests with subjects show a 15 to 52 percent improvement in terms of time to complete a modification. We also conducted an experiment in a realistic setting, using five modification scenarios to a knowledge base that were proposed by a third party. Our Script-Based KA tool found relevant scripts to guide the user in most situations.

The rest of the paper is organized as follows. First we introduce some background on script-based knowledge acquisition and our framework for representing knowledge. Following that, we discuss some important considerations for developing a KA script library. Then we describe each one of the analyses that we have carried out and report on their results. Finally we describe our implementation of a SBKA tool and present some results from its evaluation.

Background: Script-based Knowledge Acquisition

The script-based knowledge acquisition (SBKA) approach (Gil & Tallis 1997) was conceived to support users in completing a KBS modification. KBS modifications usually require changing several related parts of a system. Identifying all of the related portions of the system that need to be changed and determining how to change them is hard for users to figure out. Furthermore, if the modification is not completed, the KBS will be left inconsistent.

To assist users in performing all of the required changes, a KA tool needs to understand how changes in different parts of the system are related. In script-based knowledge-acquisition this is achieved by incorporating a library of *knowledge-acquisition scripts*, which represent prototypical procedures for modifying knowledge-based systems. KA scripts provide a context for relating individual changes of different parts of a KBS, and hence enabling the analysis of each change from the perspective of the overall modification.

Figure 1 shows an example of a typical procedure for modifying a KBS. In our knowledge representation framework which is EXPECT (Gil & Melz 1996; Swartout & Gil 1995; Gil 1994), a knowledge-based system includes a model of the domain and problem-solving methods for achieving goals in that domain. The domain model describes concepts, relations, and their instances. A problem solving method (or *method* for short) consist of a *capability description* that indicates the goals that the method is able to achieve, and a *body* that describes a procedure for achieving those goals. The body procedure can include *subgoal expressions* that have to be solved by other methods whose capability subsumes the posted subgoal. The method's body can also include *relational expressions* that make reference to other elements of the domain.

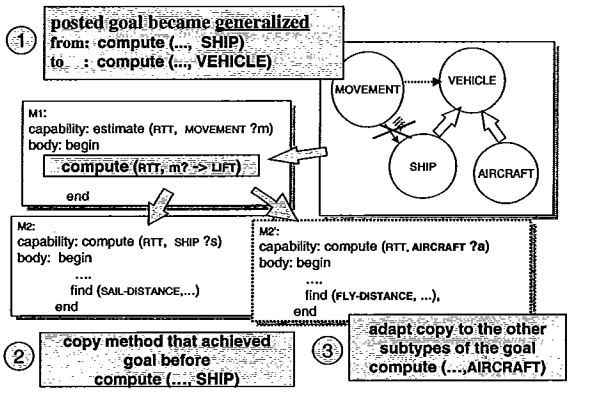

Figure 1: A typical KBS modification procedure

The domain model of Figure 1 describes two kind of VEHICLES: SHIPS and AIRCRAFT, and TRANSPORTATION MOVEMENTS with LIFTS consisting of SHIPS (crossed over). The figure also shows two problem-solving methods. Method M1 for *estimating the round-trip time (RTT) of a movement* posts a subgoal for *computing the RTT of the lift of the movement.* Because the lift of a movement consist of ships, this goal could be achieved by method M2 which *computes the RTT of ships.*

Suppose now that the LIFT relation is changed to include any kind of vehicle in its range. This change will cause the goal COMPUTE, posted by M1, to compute the RTT for both ships and aircraft. At this state, the KBS contains an error because the system will not be able to achieve the goal *compute RTT of vehicle* with any available problem-solving method. Hence, additional changes are needed to complete the modification and leave the KBS in a coherent state.

The example also describes a typical procedure for following up these kinds of changes. The procedure indicates that:

if 1) a) a posted goal is made more general (e.g., the goal *compute the round trip time of a* **ship** is changed to *compute the round trip time of a* **vehicle**) and
b) the generalized goal can now be decomposed into two disjunct subcases (e.g, the goal*compute the round trip time of a* **vehicle** can be decomposed into one case for **ship** and another case for **aircraft**), and
c) one of these subcases is equivalent to the goal before being generalized
then 2) copy the method that achieved the goal before being changed (this method can still achieve one of the subcases), and
3) adapt this copy to the other subcase (e.g., copy the method for **ships** and adapt it to **aircraft**). This way, the two methods combined would achieve the modified goal.

The following sections discuss the methods used to develop a library of KA scripts.

Developing a Library of Knowledge Acquisition Scripts

A KA scripts library is the core of a script-based knowledge-acquisition (SBKA) tool. To maximize its utility, this library should not merely be a repository of isolated KA scripts. Rather, this library should combine KA Scripts such that as a whole it covers most possible situations with minimum overlap. This section discusses some important considerations for a KA script library.

Which Procedures

One important aspect is what kind of procedures would be represented. It is useful to distinguish the following types of procedures:

1. *Macros.* Procedures for automating common sequences of changes in a knowledge-based system. The purpose of these procedures is to simplify and speed-up modifications to a KBS. An example of a procedure of this type might be a macro for splitting a problem-solving method. This would be useful, for example, when a method is too complex or when one wants to reuse a portion of it. This procedure substitutes a fragment of the original method by an invocation to a new method, and creates a new method that corresponds to the substituted code.
2. *Methods for fixing errors.* Procedures that implement common remedies to known errors types in the KB. The purpose of these procedures is to assist a naive user in fixing the error. An example might be a procedure that, to remedy the absence of a method for achieving a goal, adapts another method to that purpose. This procedure would guide a user in choosing and adapting an existing method to solve the unachieved goal.
3. *Procedures for following up changes.* Procedures for propagating the changes performed to one KB element into other related elements in the KB. The purpose of these procedures is to help a user to deal with the complexity of the interactions among elements of a KBS. An example of a procedure of this type might be a procedure that modifies a goal statement to conform to a change of the capability of the method that achieves it.

These categories are not mutually exclusive. Some KB modification procedures may belong to more than one of these classes. For example, a macro can be used to fix an error, and this fix might consist in following up a previous change. We decided to focus on the procedures for following up changes because they are more directly related to the issue of assisting users in performing all required changes of a KBS modification.

What changes

Changes can be described at different levels, from a purely syntactical level (e.g., change a goal argument from X to Y) to a knowledge level (e.g., modify a problem-solving method so instead of considering trips exclusively with ships, it now considers aircraft too).

While a low-level description would make it easier to describe and formulate changes, it fails to capture the intention behind each change, and KA Scripts for following them up end up being so general that they are typically not very useful. For example, consider a KA Script that propagates a change in a goal argument to the method that is supposed to achieve that goal. This KA script cannot provide too much guidance in modifying that method because this modification would be very different depending on the specifics of the change in the goal (e.g., whether the goal was made more general or more specific). Another drawback of using low level descriptions is that descriptions at this level might constitute only a small fraction of the change that has to be followed up. For example, a goal expression might have changed several parameters, and it would be better to follow this whole change at once and not each change to a parameter in isolation.

Therefore, we believe it is more useful to describe changes at a more conceptual level. However, we have found difficulty in systematically enumerating changes at this higher level, while enumerating syntactic changes is a relatively easy task.

We have found that the procedures for following changes are more dependent on the effect of the changes than on the change itself. First, because changes to KB elements have different effects depending on how these elements interact with each other. For example, a change to generalize the range of a relation will have different effects whether the elements in this range were used as arguments of a goal or as a domain for a relational expression. In the case of the goal, this change will cause a generalization of the goal. In the case of the relational expression, this change will cause a generalization of the domain of the relational expression. Depending on the case, this change should have to be followed up differently. In the goal case, the continuation might change the method that used to achieve that goal, while in the case of the relational expression the continuation might change the definition of the relation. Another reason to follow effects instead of changes is that different changes may produce the same effect and this effect is followed up independently of its cause (i.e., the original change). Therefore, our KA scripts were designed to follow up possible effects of changes rather than the performed changes. The following sections describe the analyses that we carried out in order to develop a KA script library.

Analysis of Syntactic Changes

Our first analysis looked at the kinds of syntactic changes that can be done to a knowledge-based system, their possible consequences, and the successive changes that could follow up on those consequences. 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 set of all possible changes was generated from the grammar of the language used to describe the knowledge bases.

The result of this analysis was a complete set of KA scripts for following up all possible 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 was too general to be very useful. The main problem was that they do not make good use of the context available, like more specific characteristics of the change (e.g., the parameter was changed to a more general type), existing knowledge (e.g., the modified goal can be decomposed into two subcases), or the changes performed to other parts of the knowledge base (e.g., a similar change was performed to a related element).

Another problem of this approach was that it produced somewhat cumbersome and redundant KA scripts. As we explained in the previous section, changes to KBS elements have different effects depending on how these elements interact with each other, and we have found that the procedures for following changes are more dependent on the effect of the changes than on the change itself. Especially because different changes can produce the same effect and this effect is followed up independently of its cause. By structuring KA Scripts around the type of change to be followed up, each KA script had to include provisions for every possible consequence that the change to be followed up can have. However, because these consequences are independent of the change itself, these same provisions have to be repeated in every related KA script.

Nevertheless, this analysis allowed us to follow a systematic procedure for enumerating KA Scripts and generate a complete set.

Analysis of KBS modification scenarios

This analysis was aimed to generate KA Scripts that were less general (and thus more helpful) than those generated in our previous analysis. Our hypothesis was that a detailed analysis of KBS modification scenarios would allow us to identify patterns of related changes more specific to the context in which those changes were performed. For this analysis, we compared a number of subsequent versions of KBSs that were saved by users as they were developing them. Another possibility for doing this analysis is to record the changes while users are editing the KB. The problem with that approach is that the analysis would include changes that are later undone by users, or partially undone and then completed in a way that has more of an error recovery flavor and where it would be best if the user went back to the original KBS and started the change again. We generated our data by comparing different versions of KBSs and reconstructed the changes done across versions.

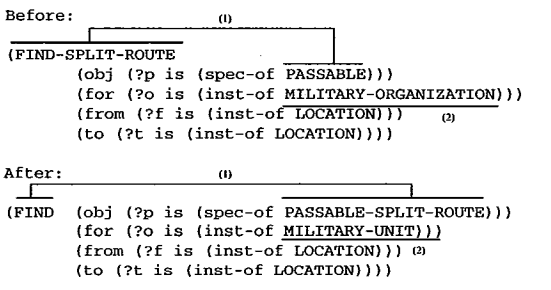

Figure 2: Two independent clusters of related changes between two versions of a method capability description. Cluster 1) corresponds to a *rephrasing of a capability* to enhance readability, while cluster 2) corresponds to a *specialization of a capability* (military-unit is a subtype of military-organization)

To reconstruct the changes performed between versions A and B of the same KBS and to identify KA scripts, we followed the following procedure:

1. **Correlating KB versions:** Build the trees of method invocation for A and B and correlate their nodes. The correlation between nodes helps in recognizing methods that have changed their names or capability descriptions between versions, and also new methods that are similar to others that existed before.
2. **Comparing versions:** For each method in the KB, find the differences between the two versions of the KBS.
3. **Identifying conceptual changes:** Not all changes performed in a method share their purpose. For each method in the KB, hypothesize the purpose of each observed change and then cluster the changes with related purposes. Figure 2 shows an example of clusters. Each cluster would constitute a *conceptual change.*
4. **Relating conceptual changes:** Find sets of related conceptual changes. One way to accomplish this is by hypothesizing what changes should have been necessary to follow up an observed conceptual change and then trying to locate them. Use the tree of method invocation to find out relations among methods.
5. **Generalizing sequences of conceptual changes:** Generalize the observed sequences of changes and determine the features from the scenario that would make these sequences possible and meaningful.
6. **Proposing other sequences:** Propose other sequences of changes by permuting the order of the changes in the sequence.

We carried out this analysis based on the following input data:

- Case Study I: 4 successive versions of a *trafficability* KBS where the user was developing an initial prototype.
- Case Study II: 6 successive versions of an *air campaign plan evaluation* KBS where the user was extending an already implemented prototype.

Case study I: Initial prototype implementation

In this scenario we identified 41 conceptual changes. The following is a list of the different conceptual changes observed and the number of times that they occurred.

1. **rephrase method capability.** The capability of a method has been rephrased by renaming, adding or deleting constant terms, usually to enhance readability. However, the method's procedure remains the same. (13 occurrences)
2. **rephrase goal.** Like the previous conceptual change but with goals (13 occurrences)
3. **restrict applicability of a method.** A method capability is specialized but the method's procedure is not changed. (9 occurrences)
4. **add exception.** A fragment of the method's procedure is embedded inside the *then* (or *else*) clause of an *If* statement. (1 occurrence)
5. **restrict result type.** Specialize result type declaration. (1 occurrence)
6. **pass an additional argument in a method invocation.** (1 occurrence)
7. **require an additional parameter in a method capability.** (1 occurrence)
8. **split a method.** Extract a fragment from a method into a newly created method. (1 occurrence)
9. **reorganize access path to domain elements.** Reorganize the sequence of domain relations used to retrieve domain elements from the KB (probably because the domain model has been reorganized too). (1 occurrence)

The following relations between conceptual changes were observed:

1. rephrase goal / rephrase capability of invoked method
2. rephrase capability / rephrase internal goals that refer to the changed capability parameters
3. restrict applicability of a method / restrict result type of method
4. restrict applicability of a method / restrict applicability of the methods invoked by internal goals that refer to the specialized parameter (i.e., propagates capability restrictions to other methods)
5. rename concept in domain model / rename concept reference in methods
6. reorganize domain model / reorganize access path to domain elements
7. rephrase goal / rephrase similar goals in same method
8. change body expression / replicate change in similar expressions in same method
9. pass an additional argument in a method invocation / require an additional parameter in a method capability

Case study II: Extending implemented prototype

In this scenario we identified 25 conceptual changes. The following is a list of the kinds of conceptual changes observed and the number of times that they occurred.

1. **rename a relational expression.** Rename a relational expression because the referred relation was also renamed in the domain model. (1 occurrence)
2. **add goal to a set of goals.** The analyzed KBS used to generate a set of goals corresponding to a set of instances. After adding an instance to the set, a new goal was automatically generated and added to the initial set of goals. (1 occurrence)
3. **create a method to achieve a new goal added to a set.** (1 occurrence)
4. **create method to achieve a new goal added to a method procedure.** (3 occurrences)
5. **reduce the number of elements from a set to be processed.** Filter the elements of a set before processing them. (1 occurrence)
6. **remove an *append* operand.** (1 occurrence)
7. **remove an unused method.** (3 occurrences).
8. **collapse two methods into one.** (2 occurrences)
9. **remove unused goal argument.** (2 occurrences).
10. **remove unused capability argument.** (2 occurrences).
11. **rephrase goal.** (2 occurrences)
12. **rephrase capability.** (2 occurrences)
13. **split a method.** (2 occurrences).
14. **regroup operations performed by a set of methods into a different set of methods.** The operations performed by a chain of methods invocations is regrouped into a different set of methods to enhance method reusability. (2 occurrences)

The following relations between conceptual changes were observed:

1. rename relation in domain model / rename relational expression
2. add goal to the set of goals / create a method to achieve a new goal added to a set
3. add goal expression to method procedure / create method to achieve new goal added to a method

4. remove goal expression from method procedure / remove unused method
5. remove unused goal argument / remove unused capability argument
6. regroup operations from a chain of methods invocations into a different set of methods / repeat the same changes for a parallel branch of method invocations.
7. rephrase goal / rephrase capability of invoked method

This scenario analysis permitted us to recognize a number of additional interesting characteristics of KBS modifications and KA scripts:

- **Conceptual changes:** Conceptual changes describe changes at a level that captures the user's intention behind the changes. For example, generalizing the type of a goal parameter conceptually correspond to *generalizing a goal.* Note that a conceptual change has several realizations. For example, another way of generalizing a goal is by generalizing its verb or by eliminating a goal parameter.

  Referring to changes at a conceptual level has the following advantages:

  - Enhance user comprehension of the KA script procedure because it is closer to the level in which users reason about changes.
  - Develop a KA Scripts library at an appropriate level of abstraction.
  - Factorize KA Scripts because referring to a conceptual change is equivalent to referring to all possible realizations of it.
  - Design a KA Script library that is not so specific to any particular knowledge-representation language because most conceptual changes have corresponding changes in other KB frameworks.

  Referring to changes at a conceptual level has the problem that it is more difficult to systematically enumerate all possible types of changes.

- **Interdependencies between KBS elements:** The observed sequences of changes allowed us to infer *Interdependencies* between elements of a KBS. For example, from two related conceptual changes, one in a method capability and the other in a method subgoal that shared the type of one of their parameters, we inferred a possible interdependency between elements of this kind. Determining the possible types of interdependencies between elements of a KBS would help us to identify new KA Scripts by analyzing how the interdependent elements should be modified in coordination. It was not possible to establish the necessary conditions for all observed interdependencies between KBS elements. Hence, some identified KA Scripts were based on the hypothetical existence of such interactions.

- **Recurrent KA scripts:** Several detected KA Scripts were observed repeatedly in different parts and in different KBSs. Whether the generation of KA scripts based on specific scenarios would produce KA scripts general enough to be reused was an issue that had concerned us before carrying out this analysis.

Although the analysis of KBS modification scenarios allowed us to identify KA Scripts that were more specific to their context, it did not help in generating a comprehensive library of KA scripts. Nevertheless, this analysis permitted us to identify important attributes of the KA Scripts that constitute the base for the third and last analysis.

Analysis of Attributes of Knowledge Acquisition Scripts

In this analysis we extended the method used in the first analysis to a full set of attributes. Starting from this set of attributes together with an enumeration of their range values, we can develop a comprehensive KA script library by defining a KA script for any feasible combination of attribute values. The attributes that we considered were as follows:

1. **Change to follow up:** We designed a typology of *conceptual changes* based on.
   (a) changed element. A KB element (or a subelement) that describes a conceptual unit (e.g., goal expression, method capability).
   (b) type of change. Whether the change created, modified, copied, or deleted an element.
   (c) transformation. The relationship between the changed element before and after the change (e.g., generalization, addition of an argument).
2. **Interdependency:** We listed all possible KB elements that could be interdependent with the changed element referred in 1 (e.g., goal / method for achieving it). A KA Script procedure would be concerned with propagating the change performed in 1 to the KB element indicated by this attribute.
3. **Strategy:** This dimension enumerates some general strategies for propagating the change indicated in 1 along the interdependency indicated in 2. For example, a strategy for propagating the addition of a new posted goal to the method that will achieve it is to generalize an existing problem-solving method that achieves a goal similar to the one recently added, so that it is applicable to both goals. We have included several strategies that either modify existing KB elements or create new KB elements based on an existing one to make it easier for users to perform the modification and to encourage knowledge reuse. The indication of which existing element should be used as a basis is the purpose of the next attribute.
4. **Base element:** This attribute enumerates good candidates to be used as a basis for strategies that modify an existing element or create a new one based on an existing one (e.g., the method that used to achieve a changed goal before starting the modification, a

method used for achieving a similar goal posted in an analogous method)

Using a set of KA script attributes to generate a KA Script library offered us some additional benefits. It allowed us to describe the content of the library by indicating the range of attribute values covered by the KA scripts in the library. It also permitted us to estimate the size of the whole library before completing its development.

An implemented KA Script Library

We have implemented a KA script library to be used with ETM (Gil & Tallis 1997), a script-based knowledge-acquisition tool that supports modifications of EXPECT knowledge-based systems (Gil & Melz 1996; Swartout & Gil 1995; Gil 1994). The following are the regions of the KA script library that we have implemented:

1. KA Scripts to follow any type of **change to a goal expressions**, that take care of its **interdependency with the problem-solving method** that achieves that goal. We considered all the strategies that consisted in creating or modifying problem-solving methods. Other possible strategies, not considered in our current implementation, include modifying the changed goal expression further to match an existing problem-solving methods in its current state.
2. KA Scripts to follow **changes to a method capability**, that take care of its **interdependency with goal expressions** to be solved by it. We considered all types of changes to method capabilities, and all strategies that consisted in creating or changing goal expressions. Other possible strategies, not considered in our current implementation, include creating or modifying further the same or other problem-solving methods to match the existing goal expressions.

These two regions of the KA Scripts space were chosen because they address the interdependencies between goal and method capability, which are in our experience one of the most prevalent and harder to solve problems that arise during knowledge-based system modifications. These two regions contain up to 100 KA Scripts altogether, from which we have implemented 37.

We were able to identify a few general operators that could be used to implement steps of the KA scripts (e.g., generalize a method, create analogous method to achieve similar goal). These operators could be reused to implement most of the steps in our KA Scripts library. For example, the operator to *Generalize a method* was used by a KA Script that in order to achieve a new goal generalizes the problem-solving method that achieves an existing goal similar to the one just added, and was also used by a KA script that, in order to achieve a goal that was generalized, generalizes the method that used to achieve that goal before being modified.

KA scripts at work

We have carried out two different studies to evaluate our KA scripts library. In the first one we conducted a series of *controlled experiments* to measure the comparative performance of subjects in modifying KBSs with and without ETM. In our second study we conducted an empirical analysis of subjects using ETM in *realistic domains and scenarios* previously unseen by the developers of ETM.

Controlled Experiments

These experiments compared the performance in modifying KBSs for subjects using ETM vs. subjects using EXPECT only. Each subject had to solve two scenarios, one of them using EXPECT and the other using ETM. One of the scenarios was slightly more complex than the other one. All of our subjects were familiar with EXPECT (but not with ETM), and had some previous exposure to the domain.

We conducted this experiment twice. The first time we only had implemented a subset of the KA script library (7 KA scripts). To put subjects in a context in which the KA scripts in the library were applicable, we indicated the first change that had to be performed for each scenario. Four subjects participated this first time. The results obtained showed that the time needed to complete the scenarios could be reduced 15% for the simpler scenario and 52% for the more complex one (Gil & Tallis 1997). 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.

We have repeated the experiment with a more complete set of KA scripts (37 KA scripts) and with no restrictions for the initial changes to the scenarios. In this occasion the time needed to complete the scenario was reduced between a 15% and a 52%. Besides, the experiment was important in that it showed that scaling up the KA scripts library did not degrade the performance of the subjects. An issue that had also concerned us.

Realistic and Unseen Scenarios

For this study we used the knowledge-bases and scenarios of the *Challenge Problems* from the *High Performance Knowledge Bases* (HPKB) DARPA project (Cohen *et al.* 1998). These problems were developed independently and with the specific purpose of testing the technologies being developed within the project. In particular, both, the domain and the modification scenarios were unknown to the designers of the KA script library before this experiment. The modification scenarios consisted in performing five extensions to a large KBS concerned with evaluating enemy workarounds for a targeted obstacle and containing 62 problem-solving methods. Each extension required creating and modifying several problem-solving methods.

In this experiment, a subject used ETM to implement the KBS extensions requested by the scenarios.

Our subject was one of the developers of the KBS who had also performed these same modifications months before the experiment without ETM. During this experiment, the subject was allowed to consult the KBS that has resulted from that previous episode, and to copy and paste fragments of that KBS in order to speed-up typing. However, we asked the subject to follow the guidance of ETM if appropriate.

We expected that our library would cover all the situations in which methods should be created or modified to follow up additions or modifications of goals. Therefore, the following analysis concentrates on those situations.

The complete modification of the KBS required the creation and modification of 16 methods. From this total, 14 were achieved with the guidance of ETM through the application of 12 KA Scripts. Most of these method modifications (12) were new methods added to the KBS, which our KA scripts helped to achieve by creating and then modifying copies of existing methods. It is interesting to note that when the subject wrongly thought that the modification was complete, ETM detected the need to perform additional changes. These changes were omitted the first time that the subject had made these modifications. KA scripts are useful not only to make changes faster but also as checklists to ensure that the changes are completed well.

Related Work

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 KA Scripts except that cliches are algorithmic fragments that are used to *generate* programs while our KA Scripts are procedures that are used to *modify* knowledge-based systems. The developers of KBEmacs have concentrated on the design of cliches and on the overall architecture to handle them, but did not directly addressed the issue of developing a library of cliches.

The Knowledge-based Software Assistant (KBSA) and its successor ARIES are other related KBSE tools (Johnson & Feather 1991; Johnson, Feather, & 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 our KA Scripts. Their main distinction lies in that evolution transformations are used to manipulate a semi-formal description of a system, while our KA scripts modified the actual implementation of a system. In developing a library of evolution transformations, the authors of these systems have followed a similar analysis to those performed by ourselves, including a detailed analysis of one scenario and an analysis of the different *semantic dimensions* embodied in software specifications. Like our KA scripts attributes, their dimensions are also independent of the notation used to describe specifications and hence can be applied to others systems too.

Conclusions

KA Scripts are a useful mechanism to guide users in modifying knowledge bases. This paper presents a number of useful insights for developing libraries of KA Scripts that are based on different kinds of analysis of modification scenarios. One of the results reported is a characterization of the modifications made to two real knowledge bases. We also describe a set of attributes that we found very useful to characterize and to systematically generate a KA Scripts library. Our initial evaluations show that our library contains KA Scripts that can be applied to most situations in real modification tasks, and that subjects spend 15 to 52 percent less time modifying knowledge bases.

Acknowledgments

We would like to thank Jose Luis Ambite, Kevin Knight, and the past and present members of the EXPECT research group. We gratefully acknowledge the support of DARPA with contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative and with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases Program.

References

Cohen, P.; Schrag, R.; Jones, E.; Pease, A.; Lin, A.; Starr, B.; Gunning, D.; and Burke, M. 1998. The Darpa High-Performance Knowledge Bases Project. *AI Magazine* 19(4).

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 Tallis, M. 1997. A Script-based Approach to Modifying Knowledge-based Systems. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence.*

Gil, Y. 1994. Knowledge Refinement in a Reflective Architecture. In *Proceedings of the Twelfth National Conference on Artificial Intelligence.*

Johnson, W. L., and Feather, M. S. 1991. Using Evolution Transformations to Construct Specifications. In *Automating Software Design.* AAAI Press. 65–92.

Johnson, W. L.; Feather, M. S.; and Harris, D. R. 1992. Representation and Presentation of Requirements Knowledge. *IEEE Transactions on Software Engineering* 18(10):853–869.

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.