Transaction-Based Knowledge Acquisition: Complex Modifications Made Easier

Yolanda Gil and Marcelo Tallis
Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

gil@isi.edu, tallis@isi.edu

In Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop, February 26-March 3, 1995. Banff, Alberta, Canada.

Abstract

Our goal is to build knowledge acquisition tools that support users in making a broad range of changes to a knowledge base, including both factual and problem-solving knowledge. These changes may require several individual modifications to various parts of the knowledge base, that need to be carefully coordinated to prevent users from introducing errors in the knowledge base. Thus, it becomes essential that our KA tools understand the consequences of each kind of change that the user may initiate, detect any harmful side-effects that can be introduced in the system, and guide the user in resolving them. To address this issue, we have developed a transaction-based approach to knowledge acquisition that can support users in making complex modifications to a knowledge base. A transaction is a sequence of changes that together modify some aspect of the behavior of a knowledge-based system, and that when only partially carried out may leave the knowledge base in an undesirable state. If a user executes a transaction partially, the knowledge acquisition tool must provide guidance to finish it and support the user in achieving the desired modification. This paper also describes our work in extending EXPECT's knowledge acquisition tool to support transaction-based mechanisms. EXPECT tracks the possible problems that arise as a consequence of each individual change to the knowledge base, keeps information about the context of each change, and uses this context to resolve the problems detected and to request the user's intervention if additional information is needed.

1 Introduction

Knowledge acquisition tools must support users in making a broad range of changes to a knowledge base in order to fully support fine-tuning a knowledge-based system's behavior. Users must be able to define new knowledge roles and modify problem-solving methods, making changes that range from small and localized modifications to complex and involved ones. These changes require several individual modifications to various parts of the knowledge base, that need to be carefully coordinated to prevent users from introducing errors in the knowledge base. Thus, it becomes essential that our KA tools understand the consequences of each kind of change that the user may initiate, detect any harmful side-effects that can be introduced in the system, and guide the user in resolving them.

We are developing the EXPECT framework [Swartout and Gil, 1995] to support users in modifying both factual and problem-solving knowledge. In EXPECT, problem-solving methods are explicitly represented and users can make fine-grained changes to their definitions. By allowing users to change the problem-solving methods within the system, all the knowledge that contributes to the behavior of the knowledge-based system can be changed. Thus, users can fully configure a knowledge-based system to conform its behavior to the requirements of the application at hand. Supporting a broad range of changes allows users to engage in complex knowledge acquisition tasks and requires mechanisms to address that complexity.

This paper presents a *transaction-based* approach to support users in making complex changes to a knowledge base. Essentially, a transaction is a sequence of small changes to a knowledge-based system that maintains the coherence and consistency of the knowledge base. A transaction-based knowledge acquisition tool supports users in completing the sequence successfully, preventing users from leaving the knowledge base in an undesirable state. Figure 1 contrasts role-limiting approaches [Marcus and McDermott, 1989, Musen, 1989, Kahn *et al.*, 1985, Klinker *et al.*, 1991, Puerta *et al.*, 1992] with our transaction-based approach. In essence, the source of expectations that drive the KA dialogue is different. In role-limiting approaches, users are expected to fill in the knowledge roles determined by the problem-solving method. In our transaction-based approach, users are expected to carry out sequences of changes that constitute legal transactions. As a result, and unlike role-limiting approaches, transaction-based KA tools do not require the problem-solving methods to be pre-defined and fixed. Instead, a transaction-based knowledge acquisition tool has pre-defined knowledge about legal transactions that allow it to follow up the consequences of individual changes to the knowledge base and prevent users from introducing undesirable errors as side effects.

The rest of the paper is organized as follows. Section 2 describes briefly the relevant aspects of the EXPECT architecture and introduces a motivating example that we use in later sections. Section 3 discusses why an analysis of the errors in a knowledge base is useful but not enough to support users in correcting these errors. Section 4 introduces transactions and their utility to support users in making complex changes to a knowledge base. Finally, we describe our initial work towards supporting knowledge base transactions in EXPECT, its limitations and our plans for future work in this area.

(a) Role-limiting approaches (b) A transaction-based approach

Figure 1: A role-limiting approach versus a transaction-based approach.

2 A Brief Overview of EXPECT

This section describes some aspects of the EXPECT architecture that are relevant to the rest of the paper. More detailed descriptions of EXPECT's explicit representations, reflective capabilities, and knowledge acquisition tool can be found in [Swartout and Gil, 1995, Gil, 1994, Gil and Paris, 1994]. We also introduce a simplified version of a Protein Synthesis domain[1] used as a source of examples throughout the paper.

2.1 EXPECT's Knowledge Bases

EXPECT's knowledge bases contain three different types of knowledge: Domain facts, domain terminology, and problem solving methods. Domain facts specify relevant data about a domain. For example, insulin is an instance of a protein. The domain terminology (or domain ontology) models objects, definitions, and the relations among them. Domain terminology and facts are represented in Loom [MacGregor, 1988, MacGregor, 1991], a state-of-the-art knowledge representation system of the KL-ONE family. These descriptions represent concepts, instances, relations, and constraints on these relations.

Figure 2 shows examples of concept definitions based on the Protein Synthesis domain that we have simplified here for clarity. A DNA structure is a molecule composed of aminoacid sequences. Genes (and vectors) are kinds of DNA structures. A resistant gene is a gene that has resistance to some substances (for example strong antibiotics). A DNA structure can have resistant genes. Restriction enzymes are a kind of enzyme that cut DNA structures. Loom uses these definitions to classify instances into the classes where they belong based on the properties of each instance.

[1]This domain models the design of gene cloning experiments and was based on the work on MOLGEN [Stefik, 1981]. These experiments use bacteria as a biological system for synthesizing a protein (e.g., insulin). The gene of the desired protein is spliced into the DNA of some bacteria, effectively altering the genetic

```
(defconcept DNAstructure
  :is (and molecule
           (at-least 1 composed-of)
           (all composed-of aminoacid-sequence)))

(defconcept gene
  :is-primitive DNAstructure)

(defconcept resistant-gene
  :is (and gene
           (at-least 1 has-resistance)
           (all has-resistance antibiotic)))

(defrelation resistance-genes
  :domain DNAstructure
  :range resistant-gene)

(defconcept restriction-enzyme
  :is (and enzyme
           (at-least 1 cuts)
           (all cuts DNAstructure)))
```

Figure 2: Some simplified descriptions in the domain of protein synthesis.

Problem solving methods are procedural descriptions for achieving goals. They consist of:

- a *capability* that represents the goal that the method can achieve, expressed with an action name and several parameters
- a *method body* that describes the procedure for achieving the method goal that is described in the capability
- a *result type* that specifies the type returned after executing the method body

Figure 3 shows several simple problem-solving methods based on the methods of our protein synthesis domain. `SPLICE` describes that in order to splice a gene into a vector, we can use a merge operation[2]. `MERGE` is a method for merging any two DNA structures by ligating them using a restriction enzyme. The method bodies shown contain subgoal expressions, but EXPECT also supports relational expressions to retrieve information from the Loom KB, conditional expressions and iteration.

EXPECT can reason about problem-solving methods in terms of what kinds of goals their capabilities can achieve, how they use domain facts in their bodies, and what kind of results they return. Let us look for example at how EXPECT reasons about goals. Notice that goals

makeup of the bacteria and causing the bacteria to produce that protein.

[2]There are other methods for splicing different types of DNA structures that are not shown here.

```
((name SPLICE)
 (capability (splice (obj (?g is (instance-of gene)))
                     (into (?v is (instance-of vector)))))
 (result-type (instance-of vector))
 (method (merge (obj ?v)
                (with ?g)))

((name MERGE)
 (capability (merge (obj (?dna1 is (instance-of DNAStructure)))
                    (with (?dna2 is (instance-of DNAStructure)))
 (result-type (instance-of DNAstructure))
 (method (ligate (obj ?dna1)
                 (with ?dna2)
                 (using (obtain (obj (instance-of restriction-enzyme))

((name OBTAIN)
 (capability (obtain (obj (?e is (instance-of enzyme)))))
 (result-type (instance-of enzyme))
 (method (select-randomly (obj ?e))
```

Figure 3: Three simplified problem solving methods in the domain of protein synthesis. They contain inaccuracies that the user needs to fix.

are represented as case frames in a case grammar, with a verb that has several slots (one of them is the direct object, denoted by `obj`). EXPECT transforms goals into Loom concepts, and uses the Loom classifier to match the method capabilities with the subgoals that arise during problem solving. For example, the subgoal in `SPLICE` for merging a vector with a gene matches the capability of `MERGE` because both vectors and genes are DNA structures.

EXPECT can be given a generic top-level goal, for example `(synthesize (obj (instance-of protein)))`. These generic goals represent the kinds of goals that the system will be given for execution, such as `(synthesize (obj insulin))`. EXPECT analyzes how to achieve this goal with the available knowledge. EXPECT expands the given top-level goal by matching it with a method and then expanding the subgoals in the method body. This process is iterated for each of the subgoals and is recorded as a search tree. Throughout this process, EXPECT propagates the types of the arguments of the top-level goal performing an elaborate form of partial evaluation supported by Loom's reasoning capabilities. During this process, EXPECT derives the interdependencies between the domain facts and the problem-solving methods. See [Swartout and Gil, 1995, Gil, 1994] for a description of how this process supports EXPECT's knowledge acquisition tool.

2.2 Modifying EXPECT's Knowledge Bases

This section describes EXPECT's basic knowledge modification commands and a motivating example. The rest of the paper shows how they are used in EXPECT to support users in carrying out the appropriate changes that will result in a new knowledge base that produces

```
((name SPLICE')
 (capability (splice (obj (?g is (instance-of gene)))
                     (into (?v is (instance-of vector)))))
 (result-type (instance-of vector))
 (method (merge (obj ?v))
                (with ?g)
                (PRESERVING (OBJ (UNION (OBJ ?G)                        [3]
                                        (WITH (RESISTANCE-GENES ?V)))))))

((name MERGE')
 (capability (merge (obj (?dna1 is (instance-of DNAStructure)))
                    (with (?dna2 is (instance-of DNAStructure)))
                    (PRESERVING ?G (SET-OF (INSTANCE-OF GENE))))        [2]
 (result-type (instance-of DNAstructure))
 (method (ligate (obj ?dna1)
                 (with ?dna2)
                 (using (obtain (obj (instance-of restriction-enzyme))
                                (PRESERVING ?G)))))                     [1]

((name OBTAIN')
 (capability (obtain (obj (?e is (instance-of enzyme))
                     (PRESERVING ?G (SET-OF (INSTANCE-OF GENE))))))     [4]
 (result-type (instance-of enzyme))
 (method (select-randomly (obj (FILTER (obj ?E)                         [5]
                                       (WITH (?E (CUTS ?G))))))))
```

Figure 4: Several modifications to the problem solving methods of Figure 3 are needed in order to protect genes from the restriction enzyme during the ligate operation. The changes are enumerated on the right side in the sequence in which they are made.

the desired behavior.

Suppose that a user of our protein synthesis domain gets an answer from the system that is not correct. The user notices an error in the method `MERGE` for merging two DNA structures. Restriction enzymes cut genes, so here the system should choose a type of enzyme that preserves the genes that are useful, in this case the genes of the protein to be synthesized and the genes of the vector that make the vector resistant to antibiotics. The following modifications need to be made (the resulting methods are shown in Figure 4):

1. add a new argument to the subgoal `obtain` in the method body of `MERGE`: `(preserving ?g)`

2. add a new parameter to the capability of `MERGE` in order to declare the variable `?g`: `(preserving (?g is (set-of (instance-of gene))))`

3. add a new argument to the subgoal `merge` in the method body of `SPLICE`, in order to pass the value of the argument `preserving`: `(preserving (obj (union (obj ?g) (with (resistance-genes ?v)))`

type	operation
modify method body	add a new expression delete an existing expression substitute an expression reorder two expressions
modify method parameters	delete a parameter add a new parameter modify an existing parameter
modify method result	substitute a method's results
delete method	delete an existing method
add new method	add a new method edited by the user add a new method by similarity with an existing method, then support the user in adapting it
instances	add a new instance add information about an instance delete an instance

Table 1: Some primitive modification operations in EXPECT's knowledge acquisition tool.

4. add a new parameter to the capability of `OBTAIN` in order to pass the value of `?g` from `MERGE`: `(preserving (?g is (set-of (instance-of gene))))`

5. change the method body of `OBTAIN` in order to use the value of `?g` by filtering out the enzymes that cut the genes: `(filter (obj ?e) (with (?e (cuts ?g)))))))`

Table 1 shows some operations that EXPECT's knowledge acquisition tool supports and that would allow a user to make these modifications to the knowledge base. Notice that the kinds of modifications that users can make are 1) broader in coverage than in role-limiting approaches, because users can change problem solving methods, 2) more fine-grained, for example users can change any expression within a method.

However, providing these modification commands is not enough to support users in refining the knowledge bases. If the user forgets to make one of the five modifications needed in our example, then the knowledge base will be left in a bad state. For example, suppose that the user forgets to add the `preserving` parameter to the capability of `OBTAIN` (i.e., the fourth of the changes listed above). The system will not be able to achieve the goal `obtain` in the new method `MERGE`': it will not match with the capability of `OBTAIN` any longer because it now has an extra parameter. The next sections describe how to provide the user with more support in 1) detecting the problem of the broken match, and 2) figuring out what modification to suggest to the user to resolve that problem.

3 Detecting and Analyzing Errors in a Knowledge Base

In addition to providing a set of commands that allow users to make fine-grained modifications to any component of the knowledge bases, EXPECT supports users further by detecting possible problems and errors that arise from these modifications and by suggesting actions to the user that can correct these problems.

The design of each component of EXPECT takes into account the possibility that the knowledge base may contain errors. The knowledge acquisition tool can then delegate to other modules the detection of the problems that an error in the knowledge bases may cause to the system when trying to solve a problem. EXPECT's problem solver is designed to detect goals that do not match with any methods and detect relations that try to retrieve information about a type of instance that is not defined in the knowledge base (e.g., retrieving the "genes" of an enzyme when only DNA structures contain genes). For example, suppose that the user does not perform the fourth of the five changes enumerated in the previous section, i.e., adding `(preserving ?g)` as a new parameter of the capability `OBTAIN`. The problem solver module would detect that `(obtain (obj (instance-of restriction-enzyme)) (preserving (set-of (instance-of gene))))` in `MERGE` cannot be achieved because it cannot be matched with any of the methods available. In addition to detecting an error, each module is able to recover from the error if possible, and to report the error's type and context in which it occurred. After detecting that a posted goal cannot be matched with any available method, EXPECT's problem solver would mark the goal as unachievable and continue problem solving expanding other goals. It would also report this error to the knowledge acquisition module together with some context information, in this case the unmatched goal with its parameters and a pointer to the node that was left unsolved.

Other modules that can detect and report errors are the parser (which detects syntax errors and undefined terms), the method analyzer (which detects errors within a method), and the instance analyzer (which detects missing information about instances). Table 2 contains the errors that these modules can currently detect.

Once the errors are detected, EXPECT can help users in fixing them as follows. For each kind of error, we analyzed the kinds of corrections to the knowledge base that users can make to solve the error. They are shown in Table 2. When EXPECT detects an error, it presents the suggested corrections as possible choices to the user. Consider the case of errors like the one above where a goal G posted by the body of a method M cannot be matched. EXPECT suggests that the user either 1) modifies the body of method M to change or to delete the expression of the goal G, 2) modifies the capability of some other method N so that it matches G, or 3) creates a new method L whose capability will match G. Each one of these suggestions would certainly resolve the error. However, not all of them will make sense in the particular context of the complex change that is in progress. In our running example, consider the situation when the user has added the new parameter `(preserving ?g)` to the goal `obtain` in `MERGE`, and EXPECT detects the error that `obtain` cannot be matched anymore. In this case, the first option presented in the table for handling this kind of error, i.e., to modify the method body to change the goal `obtain`, would certainly make the error disappear but it is probably not the one that the user will choose (since the user just modified

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

Table 2: Potential problems in the knowledge bases detected by EXPECT. Each component of EXPECT is able to detect an error, recover from the error if possible, and report the error's type and context in which it occurred. EXPECT's knowledge acquisition tool uses this information to support users in resolving errors.

the goal `obtain`.) If the system was keeping track of the sequence of changes that preceded the error then it would be able to filter and prioritize the suggestions, providing the user with better support.

Notice also that the suggestions that the system provides for each type of error are too general. The suggestions can be made more specific if more context is available. In our example, instead of recommending the user to modify the capability of `OBTAIN`, the system could realize that the reason for having to modify this capability is to match it with a goal that was added a new parameter. A more specific (and thus more helpful) suggestion is that the user adds a new parameter to the capability of `OBTAIN`, ruling out other possible modifications such as specialization of a parameter type or deleting a parameter. An even more useful suggestion would be to point out to the user what the system expects this new parameter to look like. But in order to provide this kind of specific suggestion, the system would need to keep track of more information about the context in which the error occur.

In summary, detecting errors in the knowledge bases and suggesting possible corrections is not enough. We need to provide the system with a more global view of errors and how they relate to the changes that the user previously made.

4 Transaction-Based Knowledge Acquisition

As we just argued, it is hard for a KA tool to help the user fix errors in the knowledge base unless it knows the context in which these errors were introduced. Conversely, if a KA tool allows users to make any changes to the knowledge base without following up on the side effects of each change then after a number of such modifications it becomes very hard to help the user to fix the knowledge base. Knowledge base transactions provide a framework for reasoning about the changes that a user makes in a knowledge-based system.

In the database literature, a transaction is a sequence of database accesses that if not completely executed leaves the database in an inconsistent state. For example, a transaction that transfers an amount of money x from a bank account to another account is composed of two actions: 1) decrease the amount of the source account by x, and 2) increase the amount of the destination account by x. Suppose now that we allow the system to interrupt transactions, and that a transfer of money between two accounts is interrupted after the first action is executed due to a power failure. After thousands of transactions, some completed and some incompletely executed, it becomes hard to detect what went wrong. To avoid these problems, transactions are always completely executed. Notice that even though after each individual step of the transaction the state of the database may contain inconsistencies, the database is never left in an inconsistent state at the end of a transaction.

Consider again the example presented in Section 2.2, when the user adds a new parameter to the goal `obtain` in `MERGE` but neglects to add the corresponding parameter to the capability of `OBTAIN`. The knowledge base will be left in an undesirable state because it will not be able to achieve the new goal. This suggests that the system should consider these two actions as part of a transaction.

4.1 Transactions in a Knowledge Base

A transaction is a sequence of changes that together modify some aspect of the behavior of a knowledge-based system, and that when only partially carried out may leave the knowledge base in an undesirable state. Complex transactions may be composed of smaller transactions. Transactions are ultimately built out of basic primitive transactions.

We have identified a set of primitive transactions based on an analysis of commonly occurring sequences of knowledge base modifications. The following are some examples of primitive transactions that come up in our example and the sequence of actions that they require:

T-NewVarInMethod: Introduction of a new variable in a method body

1. Modify a method body with an expression that introduces a new variable.
2. Modify the method capability to declare the new variable.

T-NewParInCap: Addition of a new parameter in a method capability

1. Modify a method capability to add a new parameter.
2. Modify the method body so that it uses the new parameter.

3. Modify the goal expressions in other method bodies that can only be matched with the original method capability and add the new parameter to those expressions.

T-NewParInGoal: Addition of a new parameter in a goal expression

1. Modify a goal expression to add a new parameter.
2. Modify the method capability that originally matched the goal expression to add the new parameter to the capability when no other method matches the new goal expression.

Transactions give us a conceptual framework to reason about how individual changes relate to one another. However, users do not think in terms of transactions. To figure out which transactions need to be executed, they have to be aware of implementation details. Hence, what we need is a framework that gives users an abstract view of the overall change they want to make, while the system recognizes and coordinates which transactions are going on. We describe next how to recognize changes in the knowledge base as part of a transaction and how to coordinate and finalize transactions once they are started.

4.2 Recognizing and Coordinating Transactions

Let us go back to our example, and see how we could support the user in making all the changes needed to preserve genes during the ligate operation. Remember that to fix this problem, the user needs to start by adding a new parameter in the `obtain` goal expression of `MERGE` to specify that these genes have to be preserved. Notice that no genes are mentioned in `MERGE`. After the user adds the parameter `(PRESERVING ?G)`, EXPECT detects the following errors:

- Error: variable `?G` in step `(PRESERVING ?G)` of the body of method `MERGE` is not declared in the capability
- Error: no method found to achieve goal `(obtain (obj (instance-of restriction-enzyme)) (preserving ?g))` in the body of method `MERGE`

We can relate these errors to the transactions we just discussed. The first error arises because the transaction T-NewVarInMethod (i.e., introduction of a new variable in a method body) was partially executed. This transaction specifies that after introducing a new variable in a method body it should be declared in the capability of the method. The second error is a consequence of the incomplete execution of the transaction T-NewParInGoal (i.e., addition of a new parameter in a goal expression of a method body). In this transaction, the addition of a new parameter to a goal expression should be followed by the addition of the corresponding parameter in the capability of the method that used to match the goal. To resolve these errors we must complete the execution of both transactions.

This example brings up several issues:

- The context in which these actions were executed determines whether a transaction was initiated or not. In our example, when the user introduces the new expression in the `MERGE` method body, this initiates the transaction T-NewVarInMethod (i.e., introduction of a new variable in a method body). This same action would not have initiated this transaction if the new expression did not contain any variables or if its variables were declared in the capability.

- Analyzing the errors in a knowledge base in isolation without the context in which they were introduced is not enough. The same error can be produced by different transactions, and hence have different solutions. For example, an error of the type "undeclared variable" is produced by two different transactions: 1) introducing a new variable in the method body and 2) deleting a parameter of a capability. In both cases, the addition of a new parameter would fix the error. However, since the parameter was just deleted by the user the second case is unlikely. A better solution in that case is to replace or remove all the references to the undeclared variable from the method body.

- Transactions are not necessarily independent. A single change can trigger several transactions. In these cases it is useful to control which transaction should be completed first. In our example, the action executed so far is the first step in both transactions. However, it is a good idea to complete the transaction T-NewVarInMethod first because this is going to give us the type of the variable `?g` which is helpful to fix the problem with the unmatched goal with the transaction T-NewParInGoal.

- Some transactions may not require complete execution if additional knowledge is available to resolve the intermediate errors. For example, the transaction that takes care of adding new parameters to a goal expression requires modifying the capability of the method that originally matched that expression. However, if the goal expression can be matched with the capability of another method in the knowledge base, then we may not need to carry out this second modification.

To address these issues, we need mechanisms for recognizing and coordinating transactions. The next section describes how EXPECT's knowledge acquisition tool approaches this problem.

5 Complex Modifications Made Easier

Based on our analysis of transactions, we created *KA scripts* by identifying groups of transactions that together modify the knowledge base in a way that is meaningful to the user. A user invokes a KA script to perform a complex modification to the knowledge base that is composed of several primitive modification operations of the kind shown in Figure 1. KA scripts isolate users from the details of primitive transactions, providing them with a global view of the overall modification. KA scripts take advantage of EXPECT's error detection capabilities, identify the transactions associated with each kind of error, and keep track of the context that is necessary to finish the execution of transactions successfully. A knowledge acquisition script detects ongoing transactions and combines them in order to coordinate the changes that they make to the knowledge base. Thus, the transactions that they contain are not necessarily executed as a sequence, instead their execution may be interleaved. A knowledge acquisition script can invoke another one that is specialized in a change needed by the first script. Notice that the sequence of changes executed by a knowledge acquisition script is also a transaction in itself, but one that modifies the knowledge base in a way that is meaningful to the user.

The knowledge acquisition scripts recognize transactions by analyzing the presence of a combination of conditions which we call *events*. These conditions are tests over the history of user's modifications which includes current and past states of the knowledge base, and the errors that arise after the execution of each modification. Events are used in EXPECT's knowledge acquisition scripts to detect unfinished transactions and to make decisions on how to guide users in completing them.

Tables 3 and 4 summarize the knowledge acquisition scripts needed to support a user in making the changes necessary for our running example. The first one supports a user in adding a parameter to a goal expression in a method body and the second one in adding a new parameter to a method capability. Notice that they invoke each other. We describe next how KA scripts work in detail through the KA script in Table 3. This script is invoked to insert the new parameter in the `obtain` goal expression of method `MERGE`. Step 1 in the script performs the insertion. Step 2 analyzes the state of the knowledge bases and detects which transactions started by looking at the errors reported and at the context. The two errors that arise in Step 2 are the ones that we presented in Section 4.2: 1) variable `?g` is not declared, and 2) `obtain` cannot be matched. However, notice that the KA script follows events, not errors. For example, `EVENT B` detects the presence of the second error (an unmatched goal) but it also detects that there was a method that matched the goal `obtain` before it was changed. We show next how the KA script resolves each event in turn.

5.1 The Missing Variable Declaration

The first error above is a result of the partial execution of the transaction T-NewVarInMethod that introduces a new variable in a method body. The missing step of the transaction is the declaration of the variable. The solution to this problem then is to add a new parameter in the capability of the method. This problem is detected by EVENT A and resolved by Step 3 of the KA script.

The information necessary to carry out an action is often not available to the KA script. The KA script always tries to infer as much information as possible. The KA script then asks the user for any other information that it cannot infer, and if this is the case it will try to present the user with its best guesses. This happens in Step 3. To add a parameter the system needs a variable name, a parameter name, and a type. In this case, the KA script infers that the variable of the new parameter is the one that needs to be declared, which is `?g` in our example. The user is then asked to provide the parameter name and type, e.g., `preserving` and `(set-of (instance-of gene))` respectively. After collecting all the information necessary, the KA script invokes the execution of another KA script that will add the parameter `(preserving (?g is (set-of (instance-of genes)))` to the capability of method

Adding a Parameter To a Goal Expression in a Method Body

```
S-AddParameterToGoalInMethod(P,G,M)

     P: Parameter to be inserted, with name Pn and argument Pa
     G: Goal expression in the body of method M where P should be inserted
     M: Method to be modified

  1 - Insert P to G in M.
      Let G' be the resulting goal expression.
  2 - Determine if this action produced any of the following events:
      EVENT A: Introduction of a new variable V in the method body of M.
      EVENT B: The goal expression G' in M and the capability D
                 in some other method N which originally matched G
                 do not match G' any longer.

  3 - When EVENT A
        {Add a new parameter Q to the capability of M to declare V}

        3.1 - Ask the user to provide the name Qn and type Qt of Q.
        3.2 - Form Q with variable V, name Pn, and type Qt.
        3.3 - S-AddParameterToMethodCapability(Q,M)

  4 - When EVENT B
        If there is another method O that matches G',
          then ask user if ok to use O instead of N to achieve G',
          else
              If user wants to define a new method for achieving G'
                then S-AddNewMethodforGoal(G')
                else
                    {Add a new parameter R to the capability D of method N}
                    4.1 - Analyze the context of M and N and generate a set of
                            plausible choices for the type Rt of R.
                    4.2 - Ask the user to provide the type Rt of R,
                            presenting as suggestions the choices in 4.1
                            and the types that are more general than those.
                            Allow user to specify a type not included in the choices.
                    4.3 - Ask the user to provide the variable name Rv of R,
                            otherwise generate a symbol for Rv.
                    4.4 - Form R with variable Rv, name Pn, and type Rt.
                    4.5 - S-AddParameterToMethodCapability(R,N)
```

Table 3: Supporting a user to add a parameter to a goal expression in a method body.

MERGE, shown in Table 4. We will not explain the execution of this KA script, but notice that in Step 3 it adds a new parameter to the goal expression merge in the method body of SPLICE to reestablish the match that existed before the capability of MERGE was changed. When the execution of that KA script is finished, the system resumes the execution of the original one. the problem of the missing variable declaration has been resolved.

Adding a Parameter To a Method Capability

```
S-AddParameterToMethodCapability(P,M)

       P: New Parameter
       M: Method to be modified

    1 - Add to capability C of M the new parameter P, with variable Pv,
           name Pn, and type Pt.
        Let C' be the resulting capability.

    2 - Determine if this action produced any of the following events:
        EVENT A: A goal expression G in some other method N
                 does not match any longer the capability C' of M.
        EVENT B: The variable Pv declared in P is not used in the body of M.

    3 - When EVENT A
          If there is another method O that matches G,
            then ask user if ok to use O instead of M to achieve G,
            else
                If user wants to define a new method for achieving G
                  then S-AddNewMethodForGoal(G)
                  else
                      {Add a new parameter to the goal expression G in method N}
                      3.1 - Ask the user to provide an expression E as the argument
                              for the new parameter. It has to be of type Pt
                              or of another type that is more specific than Pt.
                      3.2 - Let Q be the new parameter that includes Pn and E.
                      3.2 - S-AddParameterToGoalInMethod(Q,G,N)

    4 - When EVENT B
           {Make use of variable V in method body of M}
           4.1 - Ask the user to modify M to add a new step that uses V.
```

Table 4: Supporting a user to add a parameter to a method's capability.

Let us follow now how the KA script fixes the problem of the unmatched goal expression detected by `EVENT B`.

5.2 The Unmatched Goal

The second error is a result of the incomplete transaction T-NewParInGoal (i.e., adding a parameter to a goal expression). The KA script takes care of this problem in Step 4. Notice that it checks first if there is another method that matches the new goal expression of `obtain`. As we mentioned before, if this is the case then the second action of the incomplete transaction does not need to be carried out, because the error would be fixed by using the other method to achieve the new goal expression. If no other method is found, then the user can either define a new method to achieve `(obtain (obj (instance-of restriction-enzyme))` `(preserving (set-of (instance-of gene))))` or modify the capability of `OBTAIN` to add this new parameter. Suppose that the user chooses the latter.

In order to add a new parameter to the capability of `OBTAIN`, we need information about the parameter: a variable, name, and type. Again, the system tries to infer as much information as possible. Since this parameter is being added to reestablish the broken match between the new goal `obtain` and the capability of `OBTAIN`, the parameter to be added to the capability must subsume the parameter just added to the goal expression. This narrows down the choice of the parameter type to `(set-of (instance-of gene))` or a more general type. The system will present the user with these choices and ask to select one. The system also deduces that the only choice of the parameter name is `preserving`. As for the variable name, the user can provide it but the system will generate one by default. Once all this information is collected, the KA script invokes the other KA script that will add the new parameter `(preserving (?G is (set-of (instance-of gene))))` to the capability of `OBTAIN`.

The parameter added to the capability of `OBTAIN` declares a new variable which is not used in the method body. This is considered by EXPECT as an anomalous thing that needs to be fixed, and the corresponding error message will be reported. The solution to this problem – that also arises from an unfinished transaction – is to use the new variable in the method body. EXPECT suggests to the user to modify the method body with an expression that uses the new variable.

6 Discussion

If we want to allow users to change any knowledge that contributes to the behavior of a knowledge-based system, we cannot build tools that assume that some components of the knowledge base, such as problem solving mechanisms, cannot be changed. An alternative approach is to build tools that can provide support based on an understanding of the nature and the effects of the changes that users may want to make. We introduced knowledge base transactions as a framework for understanding the effects of the user's modifications. Transactions capture sequences of modifications that ensure the coherence of the knowledge base if their execution is completely carried out. We showed that individual transactions must be coordinated by higher level constructs that can reason not only about the errors that may arise as a result of a change but also about the context of the change and about previous changes made by the user. We described the implementation in EXPECT of a transaction-based approach to knowledge acquisition with knowledge acquisition scripts that detect and finish the execution of incomplete transactions. These KA scripts recognize harmful side-effects of a user's modification and guide the user in resolving them. EXPECT's KA scripts use information about the context of each knowledge base modification to provide maximum support to the user in solving the problems that may arise as the result of each modification.

One area of research that is complementary to the work presented here on detecting and resolving errors is the verification and validation of knowledge bases [O'Keefe and O'Leary, 1993]. While verification aims at ensuring that the system addresses the needs and requirements specified by the user, validation ensures that the implementation of the system functions appropriately. Most of the work on validation concentrates on rule-based systems [Ginsberg *et al.*, 1988, Loiseau, 1993]. For system validation, a suite of sample problems is given to the system to be executed, and the tool provides support for users in fixing the knowledge that produces wrong results. Recall that because EXPECT is able to analyze the solution of an abstract class of problems (as in our example, where the goal is synthesizing any type of protein), the techniques that we have developed would correspond to the validation of whole sets of problems. We would like to extend EXPECT to validate a knowledge base using also the execution of specific problems (e.g., synthesizing the insulin protein).

We plan to continue our work on KA scripts in two main fronts. One is to develop a comprehensive set of KA scripts. Since transactions provide a basic framework to design the KA scripts, it is possible to ensure that all the possible fixes of problems are considered in the KA script. Another area of future work is to design more abstract KA scripts than the ones we presented here. The ones that we have developed to date are not abstracted enough from the actual syntax of EXPECT to be accessible to end users. We plan to develop more abstract KA scripts building on the ones we currently have.

Currently, we use KA scripts as high-level commands that users can invoke for modifying the knowledge base. An alternative use of KA scripts to guide knowledge acquisition is as recognition sequences. The user would begin making changes, and the system would analyze those changes trying to recognize what kind of modification the user is trying to make. Essentially, the system would be mapping the changes that the user is making into some subsequence of a KA script. Once a KA script is recognized, the system could support the user in making individual changes ensuring that all the side effects of each change are taken care of.

Acknowledgments

We thank Bill Swartout and Bing Leng for their feedback on this work and for their efforts within the EXPECT project. We also thank Jose-Luis Ambite, Kevin Knight, and Ramesh Patil for their suggestions to improve this paper.

We gratefully acknowledge the support of the Advanced Research Projects Agency under contract no. DABT63-91-C-0025. The view and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government. Marcelo Tallis also received support from a fellowship of the Organization of American States.

References

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

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

[Ginsberg *et al.*, 1988] Allen Ginsberg, Sholom M. Weiss, and Peter Politakis. Automatic Knowledge Base Refinement for Classification Systems. *Artificial Intelligence*, 35(5):197–226, 1988.

[Kahn *et al.*, 1985] Gary Kahn, Steven Nowlan, and John McDermott. Strategies for Knowledge Acquistion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):511–522, September 1985.

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

[Loiseau, 1993] Stephane Loiseau. A method for checking and restoring the consistency of knowledge bases. *International Journal of Human-Computer Studies*, 40:425–442, 1993.

[MacGregor, 1988] R. MacGregor. A deductive pattern matcher. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, St Paul, MN, August 1988.

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

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

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

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

[O'Keefe and O'Leary, 1993] Robert M. O'Keefe and Daniel E. O'Leary. Expert System Verification and Validation: A Survey and Tutorial. *Artificial Intelligence Review*, 7:3–42, 1993.

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

[Stefik, 1981] Mark Stefik. Planning with constraints (MOLGEN: Part 1.). *Artificial Intelligence*, 16:111–140, 1981.

[Swartout and Gil, 1995] Bill Swartout and Yolanda Gil. EXPECT: Explicit Representations for Flexible Acquisition. *In Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1995.