Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition

Yolanda Gil and Eric Melz

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
email: **gil@isi.edu, melz@isi.edu**

ISI Technical Report ISI/RR-96-436

A shorter version of this paper will appear in the Proceedings of the Thirteen National Conference on Artificial Intelligence (AAAI-96), Portland, OR, August 4-8, 1996.

Abstract

Role-limiting approaches support knowledge acquisition (KA) by centering knowledge base construction on common types of tasks or domain-independent problem-solving strategies. Within a particular problem-solving strategy, domain-dependent knowledge plays specific roles. A KA tool then helps a user to fill these roles. Although role-limiting approaches are useful for guiding KA, they are limited because they only support users in filling knowledge roles that have been built in by the designers of the KA system. EXPECT takes a different approach to KA by representing problem-solving knowledge explicitly, and deriving from the current knowledge base the knowledge gaps that must be resolved by the user during KA. This paper contrasts role-limiting approaches and EXPECT's approach, using the propose-and-revise strategy as an example. EXPECT not only supports users in filling knowledge roles, but also provides support in 1) adapting the problem-solving strategy, 2) changing the types of information to be acquired about a knowledge role, 3) adding new knowledge roles, and 4) acquiring additional background information about the domain needed by the knowledge-based system. EXPECT's guidance changes as the knowledge base changes, providing a more flexible approach to knowledge acquisition. This work provides evidence supporting the need for explicit representations in building knowledge-based systems.

1 Introduction

Role-limiting approaches have been the main focus of research in knowledge acquisition (KA) tools for knowledge-based systems construction for over a decade [Birmingham and Klinker, 1993]. Several researchers have identified commonly occurring, domain-independent problem-solving strategies or inference structures that are useful for describing the reasoning behind knowledge-based systems [McDermott, 1988, Clancey, 1985, Chandrasekaran, 1986]. These problem-solving strategies determine the roles that domain-dependent knowledge plays. The task of a KA tool, then, is to guide users in filling out those roles. Several such tools have been built to support KA for a specific problem-solving strategy: SALT for propose-and-revise [Marcus and McDermott, 1989], MOLE for cover-and-differentiate [Eshelman, 1988], PROTEGE for skeletal plan refinement [Musen, 1989], etc. Each tool was designed for a specific strategy and could be used in principle to acquire knowledge for any application whose problem-solving behavior could be cast in terms of that strategy. In role-limiting approaches to KA, knowledge roles strongly determine what kinds of knowledge need to be acquired, and the dialogue with the user is centered on the acquisition of domain-dependent knowledge to fill these roles.

Although having a role-limiting strategy provides very strong guidance for knowledge acquisition, these tools lack the flexibility that knowledge-based system construction needs [Musen, 1992]. The problem-solving structure of an application cannot always be defined in domain-independent terms, as Musen explains was the case with R1 [McDermott, 1982]. Furthermore, one single problem-solving strategy may not address all of the particulars of an application, simply because it was designed with generality in mind.

More recent approaches to KA overcome these limitations by offering the system builder a library of finer-grained problem-solving strategies whose components can be used to put together a knowledge-based system [Puerta *et al.*, 1992, Runkel and Birmingham, 1993, Klinker *et al.*, 1991]. Each problem-solving strategy is then associated with a KA tool specific to that strategy. The components of the library can be designed to be as small-grained as necessary to be useful in system construction. These frameworks provide more flexibility because the overall problem-solving strategy can be customized to the needs of the application. However, their support to the user is still limited to filling knowledge roles that have been identified beforehand by the designers of these components. The kinds of modifications to the problem-solving strategy are limited to exchanging one component for another in the library. Also, a KA tool needs to be built for every problem-solving strategy.

EXPECT [Swartout and Gil, 1995, Gil, 1994, Gil and Paris, 1994] takes a different approach to knowledge acquisition. The problem-solving strategy is represented explicitly, and the knowledge acquisition tool reasons about it and dynamically derives the knowledge roles that must be filled out, as well as any other information needed for problem solving. Because the problem-solving strategy is explicitly represented, it can be modified, and as a result, the KA tool changes its interaction with the user to acquire knowledge for the new strategy. Only one KA tool needs to be built, because it can identify knowledge gaps for any problem-solving strategy that can be explicitly represented in EXPECT. EXPECT provides greater flexibility in adapting problem-solving strategies because their representations can be changed as much as needed. Because the systems that have been built to date with

EXPECT do not use a domain-independent problem-solving strategy[1], it is hard to compare role-limiting approaches with EXPECT's approach of having explicit representations to guide knowledge acquisition. This paper illustrates how EXPECT's knowledge acquisition tool works when the system is using a specific problem-solving strategy. This allows a more detailed comparison with role-limiting approaches and shows that EXPECT not only supports users in filling out knowledge roles, but extends the support to acquire additional knowledge needed for problem-solving—a process that role-limiting approaches to KA do not support.

To show how EXPECT works with a role-limiting strategy we chose propose-and-revise, one that has been the focus of much recent work within the KA community. [Schreiber and Birmingham, 1994]. Propose-and-revise was first identified as the problem-solving strategy used in VT, a system for elevator configuration [Marcus *et al.*, 1988]. Perhaps the main reason for the interest in propose-and-revise is that the VT domain has been used as a common domain for the Sisyphus effort within the KA community, where research groups are invited to show their solutions to a common problem to allow comparisons among the different frameworks [Schreiber and Birmingham, 1994]. The main sections of this paper compare EXPECT with SALT [Marcus and McDermott, 1989], the prototypical KA tool that uses a role-limiting approach for that problem-solving strategy. Contrasting EXPECT with SALT is useful to illustrate the main points of this work, but at the end of the paper we compare EXPECT with more recent approaches to building knowledge acquisition tools. Because the VT domain takes a significant amount of time to implement, we used instead a smaller domain for U-Haul ©rentals—one that also uses propose-and-revise, designed by the PROTEGE group at Stanford University, and used in their own work [Gennari *et al.*, 1993]. This domain was sufficient to allow us to implement propose-and-revise in EXPECT and to enable a more direct comparison of its KA tool with other approaches.

The paper begins by describing propose-and-revise and its use in a role-limiting tool for knowledge acquisition. Then we describe how propose-and-revise and the U-Haul domain were implemented in EXPECT. Section 4 describes EXPECT's knowledge acquisition tool. Section 5 shows several examples of how EXPECT can acquire knowledge for propose-and-revise and also support users in acquiring additional types of knowledge. We then compare EXPECT's approach with recent KA tools and approaches that have been used for propose-and-revise.

2 Role-Limiting Methods: the Case of Propose-and-Revise

This section reviews the basic propose-and-revise problem-solving strategy, and then briefly presents how SALT guides knowledge acquisition using propose-and-revise in a role-limiting approach.

[1]The problem-solving strategies used in these systems are not domain-independent. [Gil, 1994] shows examples of EXPECT's knowledge acquisition tool in a domain for transportation planning to do plan evaluation, and [Gil and Tallis, 1995] shows examples of a molecular biology domain that uses a kind of skeletal plan refinement.

2.1 Solving Configuration Design Tasks with Propose-and-Revise

Propose-and-revise is a problem-solving strategy for configuration design tasks. A configuration problem is described as a set of *parameters* or variables (input parameters have values, output parameters do not), a set of *constraints*, and a set of *fixes* to resolve constraint violations. A solution consists of a value assignment to the output parameters that does not violate any constraint.

Propose-and-revise constructs a solution by iteratively extending and revising partial solutions. The *extension* phase consists of assigning values to parameters. In the *revision* phase, constraints are checked to verify whether they are violated by the current solution and, if so, the solution is revised to resolve the violation. Violated constraints are resolved by applying fixes to the solution. A fix produces a revision of the solution by changing the value of one of the parameters that are causing the constraint violation.

Propose-and-revise was first defined as a problem-solving method for configuration in VT [Marcus *et al.*, 1988] for designing elevator systems. Input parameters for VT included features of the building where the elevator was to be installed. Output parameters included the equipment selected and its layout in the hoistway. An example of a constraint is that a model 18 machine can only be used with a 15, 20, or 25 horsepower motor. An example of a fix for a violation of this constraint is to upgrade the motor if the current configuration was using one without enough horsepower.

2.2 Knowledge Acquisition for Propose-and-Revise in a Role-Limiting Tool

SALT [Marcus and McDermott, 1989] is a knowledge acquisition tool for propose-and-revise using a role-limiting approach. In this problem-solving strategy, there are three types of knowledge roles: 1) procedures to assign a value to a parameter, which would result in a design extension, 2) constraints that could be violated in a design extension, and 3) fixes for a constraint violation. Consequently, the user can enter one of the three types of knowledge: PROCEDURE, CONSTRAINT, and FIX. For each type of knowledge, a fixed menu (or *schema*) is presented to the user (in SALT's case a domain expert) to be filled out. An example of the information provided by a user for a constraint is as follows (from [Marcus and McDermott, 1989]):

```
1  Constrained value:   CAR-JAMB-RETURN
2  Constraint type:     MAXIMUM
3  Constraint name:     MAXIMUM-CAR-JAMB-RETURN
4  Precondition:        DOOR-OPENING = SIDE
5  Procedure:           CALCULATION
6  Formula:             PANEL-WIDTH * STRINGER-QUANTITY
7  Justification:       THIS PROCEDURE IS TAKEN FROM INSTALLATION MANUAL I, P. 12b.
```

The fields to be filled in by the user include the value constrained (1), the nature of the limit that the constraint imposes on the value (2), and a procedure for specifying a value for the constraint (6). SALT includes special-purpose modules to support the user in filling up these schemas. For example, it checks that the formula is correct and that it references variables that have been defined by the user.

One great advantage of role-limiting strategies is reuse. A tool like SALT can be used to acquire knowledge in other applications that use the propose-and-revise strategy. The problem-solving strategy would be the same, thus saving a lot of effort in building a new system. Only the domain-dependent knowledge roles would need to be acquired. Parts of the KA tool would have to be adapted in the way the knowledge roles are filled (such was the experience reported when using SALT for a flow-shop scheduler [Marcus and McDermott, 1989]).

But because role-limiting approaches to KA constrain so strongly the kind of knowledge that they can acquire, they are limited in that they can only do that form of knowledge acquisition. As a knowledge acquisition tool, SALT was specifically built for a type of problem solving strategy (i.e., propose-and-revise) that used certain types of knowledge (procedure, constraint, and fix). Its interaction with the user can never change unless, of course, SALT itself is reprogrammed. As a result, SALT is not very adaptable as a knowledge acquisition tool. For example, SALT could not be used in an application domain that required using domain knowledge to select a preferred fix, because such a knowledge role does not exist in SALT's propose-and-revise strategy. The schemas cannot be changed either. For example, suppose that the user wanted to add numerical priorities to specify which constraints should be preferred over others when resolving violations. The schema for acquisition of constraints would have to be modified. Furthermore, this would require changing the implementation of propose-and-revise so that it would use this preference information in the revision phase.

Special-purpose modules are needed to acquire some specific kinds of knowledge. For example, there is a consistency checker for the formulas in the constraint schemas. The values of the input parameters are also acquired through an interface that was specifically designed for the elevator application.

SALT does not provide support in updating or maintaining the knowledge about elevator components. This would be a very useful capability, since product knowledge changes at a high rate (40-50 percent per year is the rate reported for configuration systems such as R1 and PROSE [McDermott, 1982, Wright *et al.*, 1983].)

The essence of the argument made here about SALT applies to other role-limiting KA tools such as MOLE, [Eshelman, 1988], MORE [Kahn *et al.*, 1985], and PROTEGE [Musen, 1989]. To summarize, the main limitations of role-limiting approaches to knowledge acquisition in terms of their lack of flexibility are:

- schemas cannot be changed to acquire new information about existing knowledge roles
- the problem-solving strategy is fixed and cannot be adapted or augmented
- new knowledge roles cannot be added
- the input parameter values to be acquired are fixed
- there is no support to change the domain-specific factual knowledge, e.g., about the equipment to be used in the configuration
- special-purpose modules are needed to support the acquisition of certain kinds of knowledge, e.g., the constraint's formulas

Section 5 shows how EXPECT supports the acquisition of these kinds of knowledge. We do not claim that EXPECT can support any kind of user in making these changes to the knowledge-based system. For example, domain experts will not normally have the

background knowledge to change a problem-solving strategy, which will normally require knowledge engineering skills. The focus of our work is to show that it can be done using a single KA tool that is independent of the problem-solving strategy used.

3 Explicit Representations in EXPECT

Before describing our implementation of propose-and-revise, we briefly present EXPECT's approach to representing knowledge. We will concentrate on presenting background that is directly relevant to the work presented here. More details about the architecture and knowledge acquisition tools can be found in [Gil, 1994, Swartout and Gil, 1995, Gil and Paris, 1994].

In EXPECT, both factual knowledge and problem-solving knowledge about a task are represented explicitly. This means that the system can access and reason about the representations of factual and problem-solving knowledge and about their interactions. Factual knowledge is represented in LOOM [MacGregor, 1991, MacGregor, 1988], a state-of-the-art knowledge representation system based on description logic. Every concept or class can have a definition that intensionally describes the set of objects that belong to that class. Concept descriptions can include type and number restrictions of the fillers for relations to other concepts. Relations can also have definitions. LOOM uses the definitions to produce a subsumption hierarchy that organizes all the concepts according to their class/subclass relationship. Factual knowledge includes the concept base, the instance base, and the relations among them.

Problem-solving knowledge is represented in a procedural-style language that is tightly integrated with the LOOM representations. Subgoals that arise during problem solving are solved by methods. Each method description specifies: 1) the goal that the method can achieve, 2) the type of result that the method returns, and 3) the method body containing the procedure that must be followed in order to achieve the method's goal. A method body can contain nested expressions, including subgoal expressions that need to be resolved by other methods; control expressions such as conditional statements and some forms of iteration; and relational expressions to retrieve the fillers of a relation over a concept. Some method bodies are calls to Lisp functions that are executed without further subgoaling.

We will give examples of EXPECT's representations using propose-and-revise as a strategy for solving the following type of problems in the U-Haul domain: Given the total volume that the client needs to move, the system recommends which piece of equipment (e.g., a truck, a trailer, etc.) the client should rent.

Figure 1 graphically shows parts of the factual domain model for propose-and-revise and for the U-Haul domain.[2] The upper part of the picture shows factual knowledge that is domain independent and can be reused for any domain. Constraint satisfaction (CS) problems are specified with a set of constraints and a state. The state is described with a set of variables. Constraints can have fixes. In a configuration problem, the state is a configuration. Some of the state variables denote components, i.e., pieces of equipment that together form a configuration. A component may have an upgrade, e.g., a more powerful

[2] By convention, we denote relation names with the prefix `r-`.

Figure 1: EXPECT's representation of some of the factual knowledge needed for propose-and-revise problems, for configuration problems, and for the U-Haul domain. Notice that the relations across these subdomains are naturally captured in the subsumption hierarchy.

model of that kind of equipment.

The lower part of the picture shows factual knowledge that is relevant only to the U-Haul domain. One of the state variables denotes the equipment that the client can rent, which can be a trailer, a rooftop, or a truck. There are several kinds of trucks, ranging from the smallest EasyMover to the largest HeftyMover.

There is a continuum between the representation of domain-dependent and domain-independent factual knowledge in EXPECT. They are represented in the same language, yet they can be defined and maintained separately. Once a U-Haul problem is specified as a kind of configuration problem, it inherits the fact that it has constraints and fixes. Trucks are not defined as having upgrades, since having upgrades is a way to look at components from the point of view of configuration problems. Instead, they are defined as configuration components, which have upgrades.

Figure 2 shows three different problem-solving methods. `REVISE-CS-STATE` is one of the methods that specifies how propose-and-revise works. To revise a CS state, we apply the fixes that are found for the constraints violated in the state. The other two specify a constraint and a fix in the U-Haul domain respectively. `CHECK-CAPACITY-CONSTRAINT` specifies that the capacity of the equipment rented cannot exceed the volume that the client needs to move[3]. `APPLY-UPGRADE-EQUIPMENT-FIX` applies the fix of upgrading the equipment being rented.

The representation of knowledge about constraints and fixes illustrates EXPECT's sep-

[3]This constraint may not be realistic since the client can make several trips but is part of the specification of the U-Haul domain as it was given to us.

```
(defmethod REVISE-CS-STATE
 "To revise a CS state, apply the fixes found for the constraints
  violated in the state.  The result is a CS state."
  :goal         (revise (obj (?state is (inst-of cs-state))))
  :result       (inst-of cs-state)
  :method-body  (apply
                  (obj (find (obj (set-of (spec-of fix)))
                             (for (find
                                    (obj (set-of (spec-of violated-constraint)))
                                    (in ?state)))))
                  (to ?state)))

(defmethod CHECK-CAPACITY-CONSTRAINT
  "To check the Capacity Constraint of a U-Haul configuration, check that the
   capacity of the rented equipment is smaller or equal than the volume to move."
  :goal         (check (obj CapacityConstraint)
                       (in (?c is (inst-of uhaul-configuration))))
  :result       (inst-of boolean)
  :method-body  (is-smaller-or-equal
                  (obj (r-capacity (r-rented-equipment ?c)))
                  (than (r-volume-to-move ?c))))

(defmethod APPLY-UPGRADE-EQUIPMENT-FIX
  "To apply the Upgrade Equiment Fix in a U-Haul configuration, upgrade
   the rented equipment variable.  The result is a new configuration."
  :goal         (apply (obj UpgradeEquipmentFix)
                       (to (?c is (inst-of uhaul-configuration))))
  :result       (inst-of uhaul-configuration)
  :method-body  (upgrade (obj (spec-of rented-equipment-variable))
                         (in ?c)))
```

Figure 2: Problem-solving knowledge in EXPECT.

arate representations for procedural and factual knowledge. `UpgradeEquipmentFix` is a fix for the `CapacityConstraint` regardless of how the constraint is checked or how the fix is applied. Thus, this is represented as factual knowledge as a relation between an instance of constraint and an instance of fix, as shown in Figure 1. The procedures to check the constraint and to apply the fix are part of the problem-solving knowledge base, and are executed as part of the process to revise configurations. Factual knowledge and problem-solving knowledge about constraints and fixes are related because the methods refer to the instances `CapacityConstraint` and `UpgradeEquipmentFix` in their arguments. This separation of different kinds of knowledge is key in supporting knowledge acquisition in EXPECT, as will be shown in Section 5.

Notice that, like factual knowledge, problem-solving knowledge can be domain dependent or domain independent. EXPECT uses the same language to represent both.

4 Knowledge Acquisition in EXPECT

This section briefly summarizes how EXPECT uses the explicit representations of factual and problem-solving knowledge to detect knowledge gaps and to guide knowledge acquisition. More details about EXPECT's knowledge acquisition tool can be found in [Gil, 1994, Gil and Paris, 1994]. Section 5 will illustrate how EXPECT's KA tool acquires knowledge for the propose-and-revise and the U-Haul knowledge bases.

EXPECT guides KA by requesting users to resolve errors or knowledge gaps that it detects in the knowledge bases. EXPECT's problem-solver is designed to detect these errors and to report them to the KA tool together with detailed information about how they were detected. The KA tool uses this information to support the user in fixing them. 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 problem-solving method), and the instance analyzer (which detects missing information about instances).

EXPECT's problem-solver can analyze how the different pieces of knowledge in the knowledge-based system interact. For this analysis, it takes a generic top-level goal representing the kinds of goals that the system will be given for execution. In the U-Haul example, the top-level generic goal would be `(solve (obj (inst-of uhaul-problem)))`, and a specific goal for execution would be `(solve (obj jones-uhaul-problem))`. 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 different components of its knowledge bases. This analysis is done every time the knowledge base changes, so that EXPECT can rederive these interdependencies.

The design of each module of EXPECT takes into account the possibility that the knowledge base may contain errors or knowledge gaps. For example, EXPECT's problem solver is designed to detect goals that do not match any methods, and to detect relations that try to retrieve information about a type of instance that is not defined in the knowledge base (e.g., retrieving the upgrade of a fix when only components have upgrades). 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 the context in which it occurred. For example, after detecting that a posted goal cannot match any available method, EXPECT's problem solver would mark the goal as unachievable and continue problem solving by 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 part of the problem-solving trace where the subgoal was left unsolved.

Once the errors are detected, EXPECT can help users to fix them as follows. EXPECT has an explicit representation of types of errors, together with the kinds of corrections to the knowledge base that users can make in order to solve them. This representation is based on typical error situations that we identified by hand. Table 1 shows some of the errors that can currently be detected by two of the modules: the problem solver and the instance

code	*error or potential problem*	*source*	*suggested corrections*
e1	no method found to achieve goal G in the body of method M	problem solver	modify method body modify another method's goal add a new method modify instance, concept, or relation
e2	role R undefined for type C in method M	problem solver	modify method M add relation R for type C
e3	expression E in method M has invalid arguments	problem solver	modify method M modify another method's goal modify instance, concept, or relation
e4	expression E in method M has invalid result	problem solver	modify method M modify another method's goal modify instance, concept, or relation
e5	missing filler of role R of instance I needed in method M	instance analyzer	add information about instance modify method body delete instance
e6	type of instance I is not specific enough as needed in method M	instance analyzer	specialize instance type modify method body delete instance

Table 1: Some of the 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. Underlined error codes are used in the examples.

analyzer. The error codes shown underlined will be used in the examples in the next section. When EXPECT detects an error, it presents the suggested corrections as possible choices to the user. Consider the case of error e1, where a goal G in the body of method M cannot be achieved by any method. EXPECT suggests that the user either 1) modify the body of method M to change or delete the expression of the goal G, 2) modify the goal of some other method N so that it matches G, 3) create a new method L whose goal will match G, or 4) modify an instance, a concept, or a relation that is used in G. We will see in the next section that the user's choice of one suggestion over the others depends on the context.

5 Knowledge Acquisition for Propose-and-Revise in EXPECT

In Section 2.2, we pointed out some of SALT's limitations in terms of its lack of flexibility as a knowledge acquisition tool. In this section, we illustrate how EXPECT's explicit representations support a more flexible approach to knowledge acquisition.

5.1 Acquiring Domain-Specific Factual Knowledge

Suppose that U-Haul decided to begin renting a new kind of truck called MightyMover. The user would add a new subclass of truck, and EXPECT would immediately request the

following:

E1—I need to know the capacity of a `MightyMover`.

The reason for this request is that EXPECT has detected that the capacity of rental equipment is a role that is used during the course of problem solving, specifically while achieving the goal of checking the `CapacityConstraint` with the method shown in Figure 2. This corresponds to errors of type `e5` in Table 1. While many roles may have been defined for the class `truck` (such as make and year), EXPECT will only request the information that is actually needed for problem solving. The next section illustrates that if any problem-solving method changes and other information is used, then EXPECT will request information according to the changes made.

SALT does not provide support in acquiring this kind of domain-specific factual knowledge. As we mentioned in Section 2.2, this capability would be very useful to maintain product knowledge in configuration systems.

5.2 Acquiring New Constraints and Fixes

Section 2.2 showed that SALT needs to be given definitions of schemas to enter constraints and fixes. EXPECT does not need to be given such schemas. Instead, the information in the schemas is naturally requested by EXPECT as constraints and fixes are defined by the user.

Suppose for example that the user wants to add a new constraint that restricts the rental of trailers to clients with cars made after 1990 only. The user would add a new instance of constraint: `TrailersForNewCarsOnly`. EXPECT would analyze the implications of this change in its knowledge base and signal the following problem:

E2—I do not know how to achieve the goal `(check (obj TrailersForNewCarsOnly) (in (inst-of uhaul-configuration)))`.

This is because during problem solving EXPECT calls a method that tries to find the violated constraints of a configuration by checking each of the instances of constraint of U-Haul problems. This is a case of an error of type `e1`. Before defining this new instance of constraint, the only subgoal posted was `(check (obj capacityConstraint) (in (inst-of uhaul-configuration)))` and now it also posts the subgoal `(check (obj TrailersFor NewCarsOnly) (in (inst-of uhaul-configuration)))`. There is a method to achieve the former subgoal (shown in Figure 2), but there is no method to achieve the latter.

Notice that with this error EXPECT detects that the addition of a new constraint to the factual knowledge base requires adding a new method to the problem-solving knowledge base. This illustrates how EXPECT understands the interdependencies between factual and problem-solving knowledge and uses this to guide knowledge acquisition.

To resolve **E2**, the user chooses the third suggestion for errors of type `e1` and defines the following method to check the constraint:

```
(defmethod CHECK-TRAILERSFORNEWCARSONLY-CONSTRAINT
  :goal         (check (obj TrailersForNewCarsOnly)
                       (in (?c is (inst-of uhaul-configuration))))
  :result       (inst-of boolean)
  :method-body  (is-greater-or-equal
                    (obj (r-year (r-car ?c)))
                    (than 1990)))
```

Once this method is defined, **E2** is no longer a problem and disappears from the agenda.

Notice that with this method EXPECT is acquiring information about a knowledge role. The new method corresponds to acquiring in SALT the field `formula` in the schema for the constraint knowledge role (shown in Section 2.2).

EXPECT's error detection mechanism also notices possible problems in the formula used to check the constraint. In SALT, these problems are detected by special-purpose code that checked the validity of formulas. For example, if `r-year` had not been defined EXPECT would signal the following problem (of type `e2`):

> **E3**—I do not know what is the year of a car.

When the user defines the role `r-year` for the concept `car` this error will go away. EXPECT can also detect other types of errors in the formulas to check constraints. For example, if `r-year` was defined to have a string as a range, then EXPECT would detect a problem. It would notice that there is no method to check if a string is greater or equal than a number, because the parameters of the method for calculating `is-greater-or-equal` must be numbers). EXPECT would then tell the user:

> **E4**—I do not know how to achieve the goal `(is-greater-or-equal (obj (inst-of string)) (than 1990))`.

Like **E2**, **E4** is an error of type `e1`. But in this case the user chooses a different way of resolving the error, namely to modify the definition of the relation `r-year`.

If the user defined a fix associated with the new constraint, then EXPECT would follow a similar reasoning and signal the need to define a method to apply the new fix.

We pointed out in Section 5.1 that EXPECT changes its requests for factual information according to changes in the problem-solving methods. This can be illustrated in this example of adding a new constraint. An effect of the fact that the user defined the new method to check the constraint is that new factual knowledge about the domain is needed. In particular, EXPECT detects that it is now important to know the year of the car that the client is using (and that is part of the configuration), because it is used in this new method. The following request will be generated for any client that, like in this case Mr. Jones, needs to rent U-Haul equipment:

> **E5**—I need to know the year of the car of Jones.

This is really requiring that the information that is input to the system is complete in the sense that configuration problems can be solved. In SALT, as in many other systems,

the input information (such as client preferences or the building features) is predetermined at the time the system is defined. In EXPECT, the requirements for inputs change as the knowledge base is modified. Notice that **E5** is an error of type `e5`, and is detected by the same mechanism that was used to detect **E1**, even though they request conceptually different types of information: **E1** requests information that is relevant to the U-Haul application and **E5** requests information relevant to specific client cases.

5.3 Changing the Propose-and-Revise Strategy

We pointed out in Section 2.2 that SALT does not allow users to change the problem-solving strategy or to define new knowledge roles. This section shows how this can be done with EXPECT.

Suppose that the user wants to change the revision process of propose-and-revise to introduce priorities on what constraint violations should be resolved first. The priorities will be based on which variable is associated with each constraint.

The user would need to identify which of the problem-solving methods that express propose-and-revise in EXPECT needs to be modified. The change involves adding a new step in the method to revise CS states shown in Figure 2. The new step is a subgoal to select a constraint from the set of violated constraints. The modified method is as follows (the new step is indicated with stars on the right-hand side):

```
(defmethod REVISE-CS-STATE
  :goal         (revise (obj (?state is (inst-of cs-state))))
  :result       (inst-of cs-state)
  :method-body  (apply
                  (obj (find (obj (set-of (spec-of fix)))
                             (for
                               (select                          ; ******
                                 (obj (spec-of constraint))     ; ******
                                 (from                          ; ******
                                  (find
                                    (obj (set-of (spec-of violated-constraint)))
                                    (in ?state)))))))
                  (to ?state)))
```

EXPECT would signal the following request:

> **E6**—I do not know how to achieve the goal `(select (obj (spec-of constraint)) (from (set-of (inst-of violated-constraint))))`.

This is an error of type `e5`, and it indicates that the user has not completed the modification. The user needs to create a new method to achieve this goal as follows:

```
(defmethod SELECT-CONSTRAINT
  :goal         (select (obj (spec-of constraint))
                        (from (?vc is (set-of (inst-of violated-constraint)))))
  :result       (inst-of constraint)
  :method-body  (take (obj ?vc)
                      (with (spec-of maximum))
                      (of (r-preference
                             (r-constrained-variable ?vc)))))
```

where `r-preference` is defined as a role of variables and has a numeric range, and `r-constrained-variable` is defined as a role of constraint and has `variable` as its range. The user may also need to define a new method for the `take` subgoal if there is no such method available.

With these modifications to the knowledge base, the propose-and-revise strategy that EXPECT will follow has changed. Because the representation of the new strategy is explicit, EXPECT can reason about it and detect new knowledge gaps in its knowledge base. As a result of the modification just made, there is additional factual information needed including new information about an existing knowledge role and a new kind of knowledge role. EXPECT would then signal the following requests (both of type `e5`):

E7—I need to know the constrained variable of TrailersForNewCarsOnly.

E8—I need to know the preference of equipment-variable.

E7 is one of the fields in SALT's constraint schema shown in Section 2.2, which was called `CONSTRAINED VALUE`. **E8** is a new knowledge role centered around the notion of variables. In SALT, making the change just described to the revise strategy would have required reprogramming the tool to change the problem-solving method and to add a new schema for the new knowledge role VARIABLE that would acquire preferences for each variable or parameter.

E7 and **E8** illustrate that EXPECT has noticed that the change in the problem-solving strategy requires the user to provide new kinds of information about the factual knowledge used by the strategy. This shows that in EXPECT the acquisition of problem-solving knowledge affects the acquisition of factual knowledge. Recall that **E2** illustrated the converse.

5.4 Discussion

To summarize, we revisit SALT's limitations listed in Section 2.2 and point out which of the errors just discussed illustrate how EXPECT handles knowledge acquisition in those cases:

- acquire information about existing knowledge roles according to the current problem-solving strategy: **E2**, **E7**
- the problem-solving strategy can be adapted or augmented: **E6**
- new knowledge roles can be added: **E8**
- support the user in acquiring input parameter values: **E5**
- changing the domain-specific factual knowledge: **E1**
- support the user in acquiring specific kinds of knowledge: **E3**, **E4**

Notice that these errors are detecting conceptually different knowledge gaps, yet they may correspond to the same error type. Such is the case with **E2** and **E6** that correspond to error type `e1`, and **E1** and **E8** that correspond to error type `e5`.

Notice that SALT's schema for acquiring constraints (shown in Section 2.2) asks the user for more information than EXPECT does. For example, SALT requests information about the precondition, i.e., the situations under which the constraint should be used. SALT uses this information to determine efficiently the subset of the constraints that should be checked in each configuration. Our current version of propose-and-revise does not use knowledge

about preconditions, so this information is not requested by EXPECT. If we changed the problem-solving methods so that they check the applicability of constraints according to the precondition, EXPECT would request this information from the user. Notice for example that another field in the schema for constraint acquisition is `CONSTRAINED VALUE`, which EXPECT did not ask about (see **E7**) until the methods were changed to use this information.

Throughout the examples, we have referred to a generic user wanting to make changes to the knowledge base. This is not necessarily one user, and not necessarily the end user or domain expert. For example, the end user may only enter knowledge about clients and new trucks to rent. A more technical user would be able to modify propose-and-revise. A domain expert who does not want to change the problem-solving methods can still use EXPECT to fill up knowledge roles and populate the domain-dependent factual knowledge base. Supporting a range of users would require adding a mechanism that associates with each type of user the kinds of changes that they can make to the knowledge base and limiting the users to make only those changes. The important point is that all the changes, no matter who ends up making them, are supported by the same core knowledge acquisition tool.

6 Related Work

The previous section compared EXPECT with SALT as a prototypical representative of role-limiting approaches to KA using the propose-and-revise strategy. This section compares EXPECT with more recent work in knowledge acquisition.

Some recent approaches to KA support knowledge-based system construction by offering libraries of smaller-grained role-limiting strategies that can be composed to create the overall problem-solving strategy. Such is the approach taken in PROTEGE-II [Puerta *et al.*, 1992], DIDS [Runkel and Birmingham, 1993], and SBF [Klinker *et al.*, 1991]. Modifying a problem-solving strategy involves changing one component for another one in the library. These frameworks allow a wider range of modifications to a system than tools that use a monolithic, unchangeable problem-solving structure. However, the kinds of modifications are still limited to what the compositions of different components allow. EXPECT allows even finer-grained modifications to the problem-solving methods, by adding new substeps and defining new methods to achieve them as we showed in Section 5. The composable role-limiting approaches provide very limited support if at all to a knowledge engineer who is trying to write a new problem solving component for the library. EXPECT represents the methods in a language that the KA tool understands, so it can support the user in making these changes. In addition, EXPECT's approach requires building only one single KA tool. In fact, in working out the examples shown in this paper we did not need to change the KA tool or to add new errors to those that were already defined in EXPECT.

The PROTEGE-II and DIDS research groups participated in Sisyphus and have implemented the propose-and-revise strategy [Schreiber and Birmingham, 1994]. This allows making a more detailed comparison here.

The DIDS framework supports several interesting features that we do not currently support in EXPECT. DIDS uses the flow of control specified by the problem-solving strategy to determine in which order to acquire each type of knowledge. This order helps in coordinating the KA dialogue for each of the individual methods. We could extend EXPECT to impose

an order on the requests to the user based on the subgoal expansion sequence for a method. Another interesting feature of DIDS is that it allows incorporating special-purpose KA tools, such as CAD-based acquisition tools that allow users to specify knowledge through drawings. EXPECT does not have a mechanism to integrate such tools.

PROTEGE-II includes a suite of tools for editing ontologies. Using the ontological definitions, these tools can automatically generate customized editors that are accessible to domain experts. EXPECT currently has very limited capabilities to edit factual knowledge, and could benefit from tools that organized requests to the user using appropriate structures. The library of PROTEGE-II includes not only the problem-solving strategies but also method ontologies that describe the kinds of domain-independent knowledge used in the strategies. These method ontologies are then linked by hand to the domain-specific ontology of the application through a specific language that is used to compile the ontology links. In EXPECT, the method ontology would correspond to the upper part of Figure 1 and the domain-specific ontology to the lower part of the figure. EXPECT uses the same language to represent both, and both ontologies are linked when the domain-dependent terms are made correspond to the domain-independent ones. The KA tool can detect missing links because it can reason about the way factual knowledge interacts with problem-solving knowledge and detect when the appropriate subgoals are not invoked.

A strong emphasis in both of these approaches is knowledge reuse, i.e., using the same component of the library for different problem-solving strategies and in different applications. Reusing a component in the library would be equivalent in EXPECT to invoking a top-level goal that corresponds to a whole problem-solving strategy (for propose-and-revise that goal would be `(solve (obj (inst-of cs-problem)) (using (spec-of propose-and-revise))))`. EXPECT could benefit from representing the methods that are currently part of these system's libraries so that they could be used to bootstrap the creation of new knowledge-based systems.

TAQL is a knowledge acquisition tool for weak search methods, i.e., problem-solving strategies that are more generic than something like propose-and-revise [Yost, 1993]. TAQL is not targeted for domain experts, but for users that have programming skills. TAQL provides a language that allows users to define different kinds of problem-solving strategies. The knowledge roles that need to be filled out for these strategies are generic roles that are not dependent on the specific strategy defined but on the search framework underlying TAQL. Like EXPECT, in TAQL the KA tool is strategy-independent and can provide guidance that is based on principles that have broader application than role-limiting approaches do. Some of the errors that TAQL detects correspond to errors detected by EXPECT. For example "forgetting to design a problem space" in TAQL corresponds to an error in EXPECT that a method cannot be found to achieve a goal. Unlike TAQL, we believe that the guidance provided by EXPECT is accessible to an end-user that is trying to fill out knowledge roles.

Other work in KA that has studied problem-solving strategies (including propose-and-revise) concentrates on knowledge modeling issues [Wielinga *et al.*, 1992, Domingue *et al.*, 1993]. EXPECT's KA tool is an implemented system to support users in knowledge base refinement and maintenance.

7 Conclusion

Explicit representations of problem-solving strategies can be used to support flexible approaches to knowledge acquisition. This paper shows how this is done in the EXPECT framework, using the propose-and-revise strategy as an example. We have also compared how EXPECT supports knowledge acquisition for this strategy with a well-known tool (SALT) that was built specifically for that method. EXPECT's KA tool is able to acquire the same kinds of knowledge that a tool like SALT can acquire, as well as additional kinds of knowledge that are useful in constructing a knowledge-based system. EXPECT uses the same KA mechanisms to acquire both domain-dependent and domain-independent knowledge, and can do so for any problem-solving strategy that the user defines.

Acknowledgments

We would like to thank Sheila Coyazo, Kevin Knight, Bill Swartout, Marcelo Tallis, Milind Tambe, and Andre Valente for their comments on this paper. We would also like to thank the PROTEGE group at Stanford University, and in particular John Gennari, for creating the U-Haul domain and for making it available to us. We gratefully acknowledge the support of Defense Advanced Research Projects Agency with the contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative.

References

[Birmingham and Klinker, 1993] W. Birmingham and G. Klinker. Knowledge-acquisition tools with explicit problem-solving models. *The Knowledge Engineering Review* 8(1):5–25, 1993.

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

[Clancey, 1985] W. J. Clancey. Heuristic classification. *Artificial Intelligence* 27:289–350, 1985.

[Domingue *et al.*, 1993] J. B. Domingue, E. Motta, and S. Watt. The emerging VITAL Workbench. In N. Aussenac, et al. (eds.), *Knowledge Acquisition for Knowledge-Based Systems: Proceedings of the 1993 European Knowledge Acquisition Workshop*, Springer-Verlag, 1993.

[Eshelman, 1988] L. Eshelman. MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In S. Marcus, ed., *Automating Knowledge Acquisition for Expert Systems*, pp. 37–80. Boston: Kluwer Academic Publishers, 1988.

[Gennari *et al.*, 1993] J. H. Gennari, S. W. Tu, T. E. Rothenfluh, and M. A. Musen. Mapping methods in support of reuse. In *Proceedings of the Eighth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1994.

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

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

[Gil and Tallis, 1995] Y. Gil and M. Tallis. Transaction-Based Knowledge Acquisition: Complex Modifications made Easier. In *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1995.

[Kahn *et al.*, 1985] G. Kahn, G. Nowlan, and J. McDermott. Strategies for knowledge acquisition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7:511–522, 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.

[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.

[Marcus *et al.*, 1988] S. Marcus, J. Stout, and J. McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine* 9(1):95–112, 1988.

[McDermott, 1982] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence* 19:39–88, 1982.

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

[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.

[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.

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

[Schreiber and Birmingham, 1994] G. Schreiber and B. Birmingham, eds. *Proceedings of the Eighth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1994.

[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.

[Wielinga *et al.*, 1992] B. J. Wielinga, A. Th. Schreiber, and A. Breuker. KADS: a modelling approach to knowledge acquisition. *Knowledge Acquisition* 4(1):5–54, 1992.

[Wright *et al.*, 1983] J. R. Wright, E. S. Thompson, G. T. Vesonder, K. E. Brown, S. R. Palmer, J. I. Berman, and H. H. Moore. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T Network Systems. *AI Magazine* 14(3):69–80, 1993.

[Yost, 1993] G. R. Yost. Knowledge acquisition in Soar. *IEEE Expert*, 8(3):26–34, 1993.