In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)

Deriving Expectations to Guide Knowledge Base Creation

Jihie Kim and Yolanda Gil
Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, U.S.A.
jihie@isi.edu, gil@isi.edu

Abstract

Successful approaches to developing knowledge acquisition tools use expectations of what the user has to add or may want to add, based on how new knowledge fits within a knowledge base that already exists. When a knowledge base is first created or undergoes significant extensions and changes, these tools cannot provide much support. This paper presents an approach to creating expectations when a new knowledge base is built, and describes a knowledge acquisition tool that we implemented using this approach that supports users in creating problem-solving knowledge. As the knowledge base grows, the knowledge acquisition tool derives more frequent and more reliable expectations that result from enforcing constraints in the knowledge representation system, looking for missing pieces of knowledge in the knowledge base, and working out incrementally the inter-dependencies among the different components of the knowledge base. Our preliminary evaluations show a thirty percent time savings during knowledge acquisition. Moreover, by providing tools to support the initial phases of knowledge base development, many mistakes are detected early on and even avoided altogether. We believe that our approach contributes to improving the quality of the knowledge acquisition process and of the resulting knowledge-based systems as well.

Introduction

Knowledge acquisition (KA) is recognized as an important research area for making knowledge-based AI succeed in practice (Feigenbaum 1993). An approach that has been very effective to develop tools that acquire knowledge from users is to use *expectations* of what users have to add or may want to add next (Eriksson *et al.* 1995; Birmingham & Klinker 1993; Marcus & McDermott 1989; Davis 1979). Most of these expectations are derived from the *inter-dependencies* among the components in a knowledge-base system (KBS). EXPECT (Gil & Melz 1996; Swartout & Gil 1995) and Protégé-II (Eriksson *et al.* 1995) use dependencies between factual knowledge and problem-solving methods to find related pieces of knowledge in their KBS and create expectations from them. To give an example of these expectations, suppose that the user is building a KBS for a configuration task that finds constraint violations and then applies fixes to them. When the user defines a new constraint, the KA tool has the expectation that the user should specify possible fixes for cases where the constraint is violated, and helps the user do so. These tools can successfully build expectations because there is already a body of knowledge where the new knowledge added by the user must fit in. In the configuration example, there would be problem solving knowledge about how to solve configuration tasks (how to describe a configuration, what is a constraint, what is the relation between a constraint and a fix, how to apply a fix, etc.) However, when a new knowledge base (KB) is created (or when an existing one is significantly extended) there is little or no pre-existing knowledge in the system to draw from. How can a KA tool support the user in creating a large body of new knowledge? Are there any sources of expectations that the KA tool can exploit?

This paper describes our approach to developing KA tools that derive expectations from the KB in order to guide users during KB creation. Through an analysis of the KB creation task, we were able to detect several sources for such expectations. The expectations result from enforcing constraints in the knowledge representation system, looking for missing pieces of knowledge in the KB, and working out incrementally the inter-dependencies among the different components of the KB. As the user defines new *KB elements* (i.e., new concepts, new relations, new problem-solving knowledge), the KA tool can form increasingly more frequent and more reliable expectations. We implemented a KA tool called EMeD that uses these sources of expectations to support users in adding problem-solving knowledge. Our preliminary evaluation shows an *average time savings of 30%* to enter the new knowledge. We believe it will be even higher for users who are not experienced knowledge engineers.

The paper begins by describing why KB creation is hard. Then we present our approach, and describe the KA tool that we implemented. Finally, we show the results from our experiments with several subjects, and discuss our conclusions and directions for future work.

Creating Knowledge Bases

There are several reasons why creating a knowledge base is hard:

- **Developers have to design and create a large number of KB elements.** KB developers have to turn models and abstractions about a task domain into individual KB elements. When they are creating an individual KB element, it is hard to remember the details of all the definitions that have already been created. It is also hard to anticipate all the details of the definitions that remain to be worked out and implemented. As a result, many of these KB elements are not completely flawless from the beginning, and they tend to generate lots of errors that have unforeseen side effects. Also, until a KB element is debugged and freed from these errors, the expectations created from it may not be very reliable.
- **There are many missing pieces of knowledge at a given time.** Even if the developers understand the domain very well, it is hard to picture how all the knowledge should be expressed correctly. As some part of the knowledge is represented, there will be many missing pieces that should be completed. It is hard for KB developers to keep track of what pieces are still missing, and to take them into account as they are creating new elements.
- **It is hard to predict what pieces of knowledge are related and how.** Since there is not a working system yet, many of the relationships between the individual pieces are in the mind of the KB developer and have not been captured or correctly expressed in the KB.
- **There can be many inconsistencies among related KB elements that are newly defined.** It is hard for KB developers to detect all the possible conflicts among the definitions that they create. Often times they are detected through the painful process of trying to run the system and watching it not work at all. The debugging is done through an iterative process of running the system, failing, staring at various traces to see what is happening, and finally finding the cause for the problem.

As intelligent systems operate in real-world, large-scale knowledge intensive domains, these problems are compounded. As new technology enables the creation of knowledge bases with thousands and millions of axioms, KB developers will be faced an increasingly more unmanageable and perhaps impossible task. Consider an example from our experience with a Workarounds domain selected by DARPA as one of the challenge problems of the High-Performance Knowledge Bases program that investigates the development of large-scale knowledge based systems. The task is to estimate the delay caused to enemy forces when an obstacle is targeted by reasoning about how they could bypass, breach or improve the obstacle. After several large ontologies of terms relevant to battlespace reasoning were built (military units, engineering assets, transport vehicles, etc.), we faced the task of creating the problem solving knowledge base that used all those terms and facts to actually estimate the workaround time. We built eighty-four problem-solving methods from scratch on top of several thousand defined concepts, and it took two intense months to put together all the pieces. Figure 1 shows some examples of our methods. Each method has a *capability* that describes what goals it

```
(define-method M1
(documentation "In order to estimate the time that it takes to narrow
a gap with a bulldozer, combine the total dirt volume to be moved
and the standard earthmoving rate.")

(capability (estimate (obj (?t is (spec-of time)))
              (for (?s is (inst-of narrow-gap)))))
 (result-type (inst-of number))
 (body (divide (obj (find (obj (spec-of dirt-volume))
                     (for ?s)))
         (by (find (obj (spec-of standard-bulldozing-rate))
                    (for ?s))))))

(define-method M2
(documentation "The amount of dirt that needs to be moved in any
workaround step that involves moving dirt (such as narrowing a gap
with a bulldozer) is the value of the role earth-volume for that step.")

(capability (find (obj (?v is (spec-of dirt-volume)))
                   (for (?s is (inst-of move-earth)))))
 (result-type (inst-of number))
 (body (earth-volume ?s)))

(define-method M3
(documentation "The standard bulldozing rate for a workaround step
that involves earthmoving is the combined bulldozing rate of the doz-
ers specified as dozer-of of the step.")

(capability (find (obj (?r is (spec-of standard-bulldozing-rate)))
                   (for (?s is (inst-of bulldoze-region)))))
 (result-type (inst-of number))
 (body (find (obj (spec-of standard-bulldozing-rate))
                (of (dozer-of ?s))))
```

Figure 1: Methods in a simplified workaround generation domain.

can achieve, a *method body* that specifies the procedure to achieve that capability (including invoking subgoals to be resolved with other methods, retrieving values of roles, and combining results through control constructs such as conditional expressions), and a *result type* that specifies the kind of result that the body is expected to return (more details of their syntax are discussed below). Creating each method so that it would use appropriate terms from ontologies was our first challenge. Once created, it was hard to understand how the methods were related to each other, especially when these interdependencies result from the definitions in the ontologies. Despite the modular, hierarchical design of our system, small errors and local inconsistencies tend to blend together to produce inexplicable results making it very hard to find and to fix the source of the problems. Although some portions of the knowledge base could be examined locally by testing subproblems, we often found ourselves working all the way back to our own documentation and notes to understand what was happening in the system.

In summary, it is hard for KB developers to keep in mind all the definitions that they create and to work out their interdependencies correctly. KB developers generate and resolve many errors while they build a large body of knowledge. Our goal is to develop KA tools that help users resolve these errors and, more importantly, help them avoid making the errors in the first place.

Approach

We identified several sources of expectations that KA tools can exploit in order to guide users in creating a new knowledge base. We explain our approach in terms of the problems and examples described in the previous section.

- *Difficulty in designing and creating many KB elements ⇒ Guide the users to avoid errors and look up related KB elements.*

  First, each time a KB element is created by a user, we can check the dependencies within the element and find any potential errors based on the given representation language. For example, when undefined variables are used in method body, this will create an expectation that the user needs to define them in the method.

  In our example, Method M1 has two variables, ?t and ?s, defined in its capability, and if the method body uses a different variable, the system can send a warning message to the user. Likewise, if a concept definition says that a role can have at most one value but also at least two values, then this local inconsistency can be brought up. By isolating these local errors and filtering them out earlier in the KB development process, we can prevent them from propagating to other elements in the system.

  However small the current KB is, if there are KB elements that *could be* similar to the one being built, then they can be looked up to develop expectations on the form of new KB element. For example, developers may want to find existing KB elements that are related with particular terms or concepts based on the underlying ontology. If there is a concept hierarchy, it will be possible to retrieve KB elements that refer to superconcepts, subconcepts, or given concepts and let the user develop expectations on the current KB element based on related KB elements. For example, if a developer wants to find all the methods related to moving earth, the system can find the above methods, because narrow-gap and bulldoze-region are subtypes of move-earth. When the user adds a new method about moving earth to fill a crater, then it may be useful to take them into account. Specifically, M1 can generate expectations on how a method for estimating time to fill a crater should be built.

- *Many pieces of knowledge are missing at a given time: ⇒ Compute surface relationships among KB elements to find incomplete pieces and create expectations from them*

  The KA tool can predict relationships among the methods based on what the capability of a method can achieve and the subgoals in the bodies of other methods. For example, given the three methods in Figure 1, method M1 can use M2 and M3 for its two subgoals — find dirt volume and find bulldozing rate. These relationships can create method-submethod trees that are useful to predict how methods will be used in problem solving. In the process of building this kind of structure, the system can expose missing pieces in putting the methods together. For example, *unmatched subgoals* can be listed by collecting all the subgoals in a method that cannot be achieved by any of the already defined methods. The user will be reminded to define the missing methods and shown the subgoals that they are supposed to match. In Figure 1, if a method for the subgoal of method M3 to find the standard bulldozing rate of given dozer is not defined yet, the user is asked to define one and may create one that only works for military dozer or any dozer in general.

  Similarly, if a concept is used in a KB element definition but not defined yet, then the system will detect the *undefined concept*. Instead of simply rejecting such definition, if the developer still wants to use the term, the KA tool can collect undefined concepts and create an expectation that the developer (or other KB developers) will define the term later.

- *Difficulty in predicting what pieces of knowledge are related and how ⇒ Use surface relationships to find unused KB elements and propose potential uses of the elements*

  The above surface relationships among KB elements, such as method-submethod relationship can also help detect unused KB elements. If a method is not used by any other methods, then it can be collected into an *unused method list*. In addition to finding such unused methods, the KA tool can propose potential uses of it. For example, if the capability of a method is similar to one of the unmatched subgoals (e.g., same goal name and similar parameter types), then a potential user of the method will be the method that has the unmatched goal.

  In the same way, concepts created but not referred to in any other definitions can be collected into an *unused concept list*. The KA tool can develop expectations of KB elements that will use the definitions or perhaps even deleting these concepts if they end up being unnecessary.

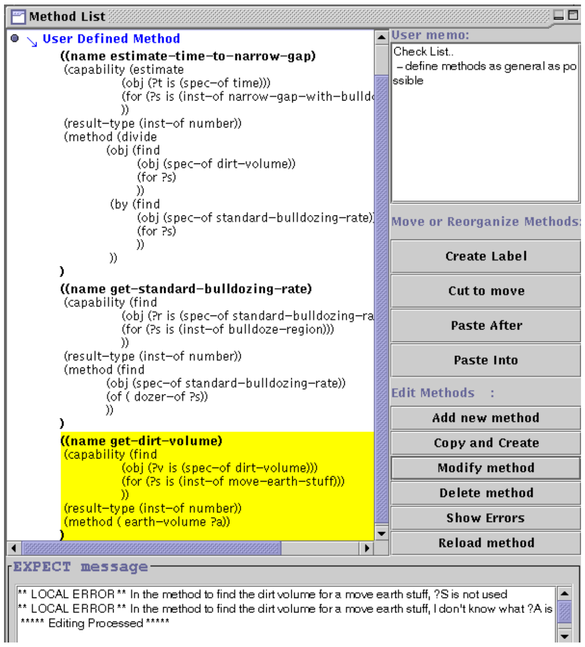
- *Inconsistencies among newly defined KB elements ⇒ Help users find them early and propose fixes*

  The KA tool can check if the user-defined result type of a method is inconsistent with what the method body returns based on the results of the subgoals. If there are inconsistent definitions, the system will develop an expectation that user has to modify either the current method or the methods that achieve the subgoals.

  Also, for concept definitions, there can be cases where a user wants to retrieve a role value of a concept, but the role is not defined for the concept. In addition to simply detecting such a problem, the system may propose to define the role for the concept or to change the method to refer to a different but related concept that does have that role.

  Finally, once the KA tool indicates that there are no errors, inconsistencies, or missing knowledge, the user can run the inference engine, exposing additional errors in solving a given problem or subproblems. The errors are caused by particular interdependencies among KB elements that arise in specific contexts. If most of the errors are detected by the above analyses, users should see significantly fewer errors at this stage.

Notice that as the KB is more complete and more error-free it becomes a stronger basis for the KA tool in creating

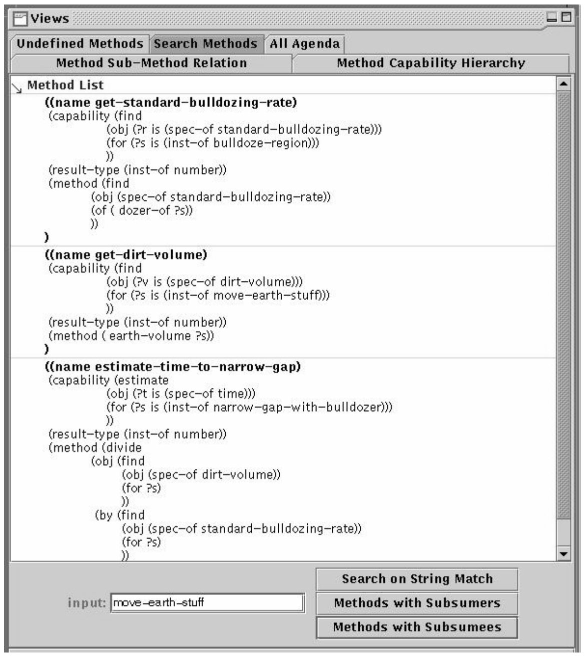


```
((name estimate-time-to-narrow-gap)
 (capability (estimate
              (obj (?t is (spec-of time)))
              (for (?s is (inst-of narrow-gap-with-bulld
              ))
 (result-type (inst-of number))
 (method (divide
          (obj (find
                (obj (spec-of dirt-volume))
                (for ?s)
                ))
          (by (find
                (obj (spec-of standard-bulldozing-rate))
                (for ?s)
                ))
          ))
)
((name get-standard-bulldozing-rate)
 (capability (find
              (obj (?r is (spec-of standard-bulldozing-ra
              (for (?s is (inst-of bulldoze-region)))
              ))
 (result-type (inst-of number))
 (method (find
          (obj (spec-of standard-bulldozing-rate))
          (of ( dozer-of ?s))
          ))
)
((name get-dirt-volume)
 (capability (find
              (obj (?v is (spec-of dirt-volume)))
              (for (?s is (inst-of move-earth-stuff)))
              ))
 (result-type (inst-of number))
 (method ( earth-volume ?a))
)
```

```
Check List.
 - define methods as general as po
ssible
```

```
** LOCAL ERROR ** In the method to find the dirt volume for a move earth stuff, ?S is not used
** LOCAL ERROR ** In the method to find the dirt volume for a move earth stuff, I don't know what ?A is
***** Editing Processed *****
```

Figure 2: EMeD Interface (Editor).



```
((name get-standard-bulldozing-rate)
 (capability (find
              (obj (?r is (spec-of standard-bulldozing-rate)))
              (for (?s is (inst-of bulldoze-region)))
              ))
 (result-type (inst-of number))
 (method (find
          (obj (spec-of standard-bulldozing-rate))
          (of ( dozer-of ?s))
          ))
)
((name get-dirt-volume)
 (capability (find
              (obj (?v is (spec-of dirt-volume)))
              (for (?s is (inst-of move-earth-stuff)))
              ))
 (result-type (inst-of number))
 (method ( earth-volume ?s))
)
((name estimate-time-to-narrow-gap)
 (capability (estimate
              (obj (?t is (spec-of time)))
              (for (?s is (inst-of narrow-gap-with-bulldozer)))
              ))
 (result-type (inst-of number))
 (method (divide
          (obj (find
                (obj (spec-of dirt-volume))
                (for ?s)
                ))
          (by (find
                (obj (spec-of standard-bulldozing-rate))
                (for ?s)
                ))
```

input: move-earth-stuff

Figure 3: Search Methods in EMeD.

expectations to guide the user.

EMeD: Expect Method Developer

We have concentrated our initial effort in developing a KA tool that uses these kinds of expectations to support users to develop problem solving methods. We built a KA tool called *EMeD* (EXPECT Method Developer) for the EXPECT framework for knowledge-based systems (Gil & Melz 1996; Gil 1994; Swartout & Gil 1995).

An EXPECT knowledge base is composed of factual knowledge and of problem-solving knowledge. The factual knowledge includes concepts, relations, and instances in Loom (Macgregor 1990), a knowledge representation system of the KL-one family. The problem-solving knowledge is represented as a set of problem-solving methods such as those shown in Figure 1. As described earlier, each method has a capability, a result type, and a method body. Within the capability and body sections, each goal is expressed as a goal name followed by a set of parameters. Also, each parameter consists of a parameter name and a type.

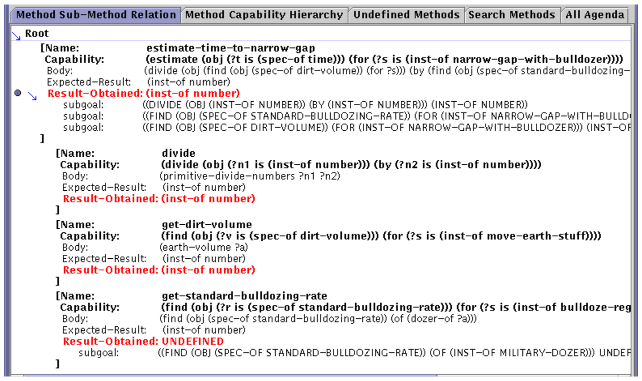
Figure 2 shows the method editor in the EMeD user interface. There is a list of current methods and buttons for editing methods. Users can add, delete, or modify the methods using these buttons. (Other buttons and windows will be explained later.) Users often create new methods that are similar to existing ones so the tool has a copy/edit facility.

Every time a new method is defined, the method is checked for possible parsing errors based on the method representation language. If there are interdependencies among the subparts of a method, they are also used in detecting errors. For example, if a variable is used but not defined for the method, the same variable is defined more than once, or there are unused variables, the system will produce warning messages. Also, if there were terms (concepts, relations, or instances) used in a method but not defined in the KB yet, error messages will be sent to the developer. When the term definition is obvious, as the verbs used in capabilities, a definition will be proposed by the tool. In Figure 2, the small panel in the bottom left corner with the label "EXPECT message" displays these errors. Using this method definition checker, users can detect the local errors earlier, separating them from other types of errors.

Users can find existing methods related with particular terms in concepts, relations or instances through the Loom ontology. The KA tool can retrieve methods that refer to subconcepts, superconcepts, or a given concept and let the user create new methods based on related methods. Figure 3 shows the result from retrieving methods about moving earth. The system was able to find all the methods in Figure 1, because narrow-gap and bulldoze-region are subconcepts of move-earth.

Figure 4 shows relationships among methods based on how the subgoals of a method can be achieved by other methods. The trees built from this are called *method sub-method relation trees*. There can be multiple trees growing in the process of building a number of methods when they are not fully connected. These method-relation trees are incomplete problem-solving trees to achieve some intermediate subgoal. The (sub)trees should be eventually put together to build a problem-solving tree for the whole problem. For example, given these three methods, the system can build a method-relation tree, as shown in Figure 4.

Method sub-method relation trees can be used to detect *undefined methods* based on the subgoals in a method that



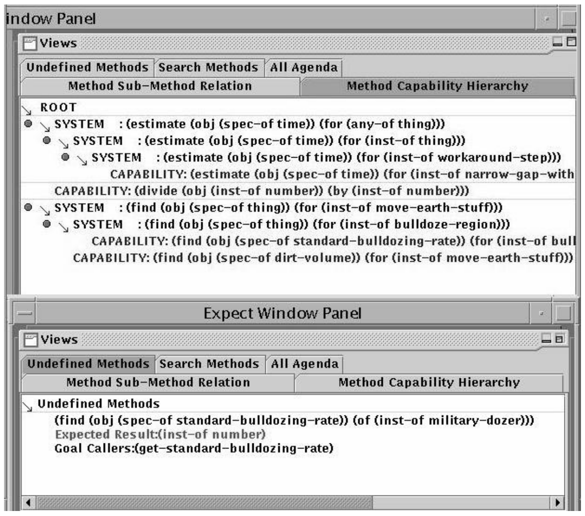
```
Root
    [Name:            estimate-time-to-narrow-gap
     Capability:      (estimate (obj (?t is (spec-of time))) (for (?s is (inst-of narrow-gap-with-bulldozer))))
     Body:            (divide (obj (find (obj (spec-of dirt-volume)) (for ?s))) (by (find (obj (spec-of standard-bulldozing-
     Expected-Result:  (inst-of number)
     Result-Obtained: (inst-of number)
       subgoal:       ((DIVIDE (OBJ (INST-OF NUMBER)) (BY (INST-OF NUMBER))) (INST-OF NUMBER))
       subgoal:       ((FIND (OBJ (SPEC-OF STANDARD-BULLDOZING-RATE)) (FOR (INST-OF NARROW-GAP-WITH-BULLD
       subgoal:       ((FIND (OBJ (SPEC-OF DIRT-VOLUME)) (FOR (INST-OF NARROW-GAP-WITH-BULLDOZER))) (INST-OF
    ]
      [Name:            divide
       Capability:      (divide (obj (?n1 is (inst-of number))) (by (?n2 is (inst-of number))))
       Body:            (primitive-divide-numbers ?n1 ?n2)
       Expected-Result:  (inst-of number)
       Result-Obtained: (inst-of number)
      ]
      [Name:            get-dirt-volume
       Capability:      (find (obj (?v is (spec-of dirt-volume))) (for (?s is (inst-of move-earth-stuff))))
       Body:            (earth-volume ?a)
       Expected-Result:  (inst-of number)
       Result-Obtained: (inst-of number)
      ]
      [Name:            get-standard-bulldozing-rate
       Capability:      (find (obj (?r is (spec-of standard-bulldozing-rate))) (for (?s is (inst-of bulldoze-reg
       Body:            (find (obj (spec-of standard-bulldozing-rate)) (of (dozer-of ?a)))
       Expected-Result:  (inst-of number)
       Result-Obtained: UNDEFINED
         subgoal:       ((FIND (OBJ (SPEC-OF STANDARD-BULLDOZING-RATE)) (OF (INST-OF MILITARY-DOZER))) UNDEF
      ]
```

Figure 4: A Method Sub-method Relation Tree.

are not achieved by any existing methods. These can be collected and users can be informed of them. If there are constraints imposed on the methods to be built, such as the expectations coming from the methods that invoke them, then these can be also incorporated. For example, the method to find the standard bulldozing rate of a step calls a subgoal to find the standard bulldozing rate of a given dozer, which is undefined yet. Since the result type of the given method is a number, the system can expect (through an interdependency analysis) the same result type for the undefined method. Figure 5 (bottom window) shows the capability that the tool proposes for the currently undefined method — a method to find standard bulldozing rate of a given military dozer.

In the process of building this method-relation tree, there can be subgoals whose parameters are not fully specified because their arguments are subgoals that are not achieved by any of existing methods. For example, given the method to estimate the time to narrow gap (the first method in Figure 1) only (i.e., if M2 and M3 were missing), its subgoal 'divide' has two parameters with parameter names 'obj' and 'by'. Because the arguments to divide are the subgoals 'find dirt-volume of the step' and 'find standard-bulldozing-rate for the step' whose methods would be undefined, the tool could not fully state the goal. This would be represented as 'divide (obj UNDEFINED) (by UNDEFINED)'. However, one of the built-in methods in EXPECT has capability of 'divide (obj Number) (by Number)', and the tool creates a link between this and the subgoal as a *potential interdependency*. Users can use this hint to make the potential interdependency a real one or create other appropriate methods.

There are other relationships among problem-solving methods based on their capabilities that the KA tool can exploit. For example, a hierarchy of the goals based on the subsumption relations of their goal names and their arguments can be created. In the hierarchy, if a goal is to build a military bridge, and another goal is to build a kind of military bridge, such as an AVLB, then the former subsumes the latter. This dependency among the goal descriptions of the methods (called *capability tree*) is useful in that it allows



```
ROOT
  SYSTEM   : (estimate (obj (spec-of time)) (for (any-of thing)))
    SYSTEM   : (estimate (obj (spec-of time)) (for (inst-of thing)))
      SYSTEM   : (estimate (obj (spec-of time)) (for (inst-of workaround-step)))
          CAPABILITY: (estimate (obj (spec-of time)) (for (inst-of narrow-gap-with
  CAPABILITY: (divide (obj (inst-of number)) (by (inst-of number)))
  SYSTEM   : (find (obj (spec-of thing)) (for (inst-of move-earth-stuff)))
    SYSTEM   : (find (obj (spec-of thing)) (for (inst-of bulldoze-region)))
        CAPABILITY: (find (obj (spec-of standard-bulldozing-rate)) (for (inst-of bull
    CAPABILITY: (find (obj (spec-of dirt-volume)) (for (inst-of move-earth-stuff)))
```

```
Undefined Methods
    (find (obj (spec-of standard-bulldozing-rate)) (of (inst-of military-dozer)))
    Expected Result:(inst-of number)
    Goal Callers:(get-standard-bulldozing-rate)
```

Figure 5: A Capability Tree and Undefined Methods.

the user to understand the kinds of sub-problems the current set of methods can solve. To make their relationships more understandable, EMeD also computes potential capabilities. For example if there are super-concepts defined for the parameters of a capability, a capability with these concepts is created as a parent of the capability. The capability tree for the given example methods is in Figure 5 (top window).

Finally, there are user-expected dependencies among the problem-solving methods, which are usually represented in comments or by grouping of methods in the files where they are built. They do not directly affect the system, but they often become the user's own instrument to understand the structure of what they are building. Also, it can be the user's own interpretation of additional interdependencies among methods. EMeD provides a way of organizing methods into hierarchies and groups, and allows users to provide documentation for the methods. In Figure 2, the "Move or Organize Methods" buttons support these functions.

In addition, EMeD can use the expectations derived from running the problem solver, detecting problems that arise while attempting to build a complete problem-solving tree.

Preliminary Evaluations

We performed a preliminary evaluation of our approach by comparing the performance of four subjects in two different KA tasks from a Workarounds domain. Each subject did one of the tasks with EMeD and the other task using a version of EMeD that only allowed them to edit methods (the buttons to add, delete, or modify methods), but did not have any additional support from EMeD. Before the experiment, each subject was given a tutorial of the tools with simpler scenarios. The scenarios and tools were used in different orders to reduce the influences from familiarity with tools or fatigue. Each experiment took several hours, including the tutorial, and we took detailed transcripts to record actions performed by the subjects. The subjects had some previous experience in building EXPECT knowledge

Results	Total Time(min)	Number of Methods Added	Time/Method (min)
Without EMeD	218	25	8.72
With EMeD	153	24	6.38

Average time savings: 30 %

Table 1: Results from experiments with subjects.

Functionality	Number of Times Used
Undefined Methods	23
Editor Error Message	17
Method Sub-method Relation Tree	7
Capability Tree	10
Search Methods	1
Method Organizer	2
Problem Solving Agenda	5

Table 2: Number of times that the different components of EMeD were used.

bases, but not with EMeD.

In Table 1, the total time is computed by summing the times with each subject for each tool. The time for each subject is the time to complete the given task (by creating a successful problem-solving tree and eliminating errors). EMeD was able to **reduce the development time to 70%** of the time that users needed without it. Subjects built a comparable number of methods with the different tools. Note that the subjects were not exposed to the EMeD environment before, but were very familiar with the EXPECT framework. The time savings may be more if the subjects had more familiarity with EMeD. We had multiple trial experiments with one of the subjects, with slightly different tasks, and the subject had become more and more skillful, reducing the time per each action. For these reasons, we expect that the time reduction with EMeD will be larger in practice. Also note that there is a practical limit to the amount of time saved using any KA tool. There is a significant amount of time that users spend doing tasks such as thinking and typing, where a machine can provide little help. We would like to measure the improvement over the time actually doing knowledge input, instead of the time to complete a KA task.

We counted the number of times each component of EMeD was used during the experiments, as shown in Table 2. The list of undefined methods was most useful (used 23 times to build 24 methods) during the experiment, and the subjects checked it almost every time they created a new method. The subjects seemed to be comparing what they expected with the list created by the KA tool, and built new methods using the suggestions proposed by the system. The error messages showed after editing each individual method effectively detected the errors within a method definition, and was used every time there were local errors in the definition.

Users looked at the Method Sub-method Relation Tree but not as many times as what we expected. The subjects felt the tree was useful but there were too many items shown for each node, making it hard to read. We are planning to display items selectively, showing the details only when they are needed. The Capability Tree was often used to find some capability description of another method needed while defining a method body. However, the hierarchical structure was not so meaningful to the subjects, since sometimes people choose arbitrary concepts (compute, estimate, etc.) to describe their capabilities. We are planning to develop a better way of organizing the methods based on what tasks they can achieve.

Search Methods and Method Organizer were used very little during the experiment. Since the size of the KB was relatively small (about 3 methods were given in the beginning and 6 methods were built for each task), the subjects were able to see them easily in the editor window. However, during real KB development, the size of the KB is usually much larger, and we expect that they will be more useful in real settings.

With EMeD, the subjects run the problem solver mostly to check if they had finished their tasks. EMeD was able to find errors earlier and provide guidance on how they may fix the problem, filtering out most of the errors. Without EMeD, the subjects run the problem solver to detect errors, and ended up spending more time to find the sources of errors and fix them.

Related Work

There are other KA tools that take advantage of relationships among KB elements to derive expectations. Teireisias (Davis 1979) uses *rule models* to capture relationships among the rules based on their general structure and guide the user in following that general structure when a new rule is added that fits a model. Some of the capabilities of EMeD are similar in spirit to the rule model (e.g., the method capability tree), and are also used in EMeD to help developers understand potential dependencies among the KB elements. Other KA tools (Eriksson *et al.* 1995; Birmingham & Klinker 1993; Marcus & McDermott 1989) also use dependencies between factual knowledge and problem-solving methods to guide users during knowledge acquisition. These tools help users to populate and extend a system that already has a significant body of knowledge, but they are not designed to help users in the initial stages of KB development. More importantly, these tools are built to acquire factual domain knowledge and assume that users cannot change or add problem-solving knowledge. In this sense, EMeD is unique because it guides users in adding new problem-solving methods.

In the field of software engineering, it has been recognized that it is generally better to focus on improving the process of software development rather than on the output program itself (Dunn 1994). Our approach embraces this view and tries to improve the initial phases of KB development. Some previous work on using formal languages to specify knowledge bases (Fensel, Angele, & Studer 1998) is inspired by software engineering approaches. This work provides a framework for users to model and capture the initial requirements of the system, and require that users are experienced with formal logic. Our approach is complementary in that it addresses the stage of implementing the knowledge-based system, and we believe that our formalism is more accessible to users that have no formal training. Other approaches (Wielinga, Schreiber, & Breuker 1992; Gaines & Shaw 1993) support users in the initial stages of development by providing a methodology that can be followed systematically to elicit knowledge from experts and to design the new system. These methodologies can be used in combination with our approach.

There is also related research in developing tools to help users build ontologies (Fikes, Farquhar, & Rice 1997; Terveen & Wroblewski 1990; Clark & Porter 1997). Unlike our work, these tools do not tackle the issue of using these ontologies within a problem-solving context. Many of the research contributions in these tools concern the reuse of ontologies for new problems, collaborative issues in developing knowledge bases, and the visualization of large ontologies. We believe that integrating our approach with these capabilities will result in improved environments to support KB creation.

Conclusion

We analyzed the process of KB development to support KB creation and KB extension, and found a set of expectations to help KA tools guide users during the development process. We have classified the sources of errors in the KB development process based on their characteristics, and found ways to prevent, detect, and fix errors earlier. These expectations were derived from the dependencies among KB elements. Although EMeD aims to provide support for KB creation, its functionality is also useful for modifying existing knowledge or populating a KB with instances.

We are now extending the EMeD framework to be able to derive expectations in solving particular problems. Currently EMeD computes relationship among the KB components regardless of the context. Depending on what problem episode we are solving, the relationships may show different patterns, since the problem-solving methods may become specialized.

In our initial evaluations, EMeD was able to provide useful guidance to users reducing KB development time by 30%. We expect that EMeD will be even more beneficial for domain experts who don't have much KA experience. EMeD also opens the door to collaborative tools for knowledge acquisition, because it captures what KA tasks remain to be done and that may be done by other users.

Acknowledgments

We gratefully acknowledge the support of DARPA with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases Program and with contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative. We would like to thank the members of the EXPECT project, Andre Valente, Jim Blythe, Marcelo Tallis, Surya Ramachandran, and Bill Swartout for their thoughtful input on this work. We also would like to thank Kevin Knight for the helpful comments on earlier drafts.

References

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

Clark, P., and Porter, B. 1997. Building concept representations from reusable components. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 369–376.

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

Dunn, R. H. 1994. Quality assurance, Encyclopedia of software engineering. 2.

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

Feigenbaum, E. 1993. The tiger in the cage. A Plenary Talk in Fifth Innovative Applications of AI Conference.

Fensel, D.; Angele, J.; and Studer, R. 1998. The knowledge acquisition and representation language KARL. *IEEE Transactions on Knowledge and Data Engineering* 10(4):527–550.

Fikes, R.; Farquhar, A.; and Rice, J. 1997. Tools for assembling modular ontologies in ontolingua. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 436–441.

Gaines, B. R., and Shaw, M. 1993. Knowledge acquisition tools based on personal construct psychology. *The Knowledge Engineering Review* 8(1):49–85.

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

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

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

Swartout, W., and Gil, Y. 1995. EXPECT: Explicit representations for flexible acquisition. In *Proceedings of the Ninth Knowledge-Acquisition for Knowledge-Based Systems Workshop*.

Terveen, L. G., and Wroblewski, D. 1990. A collaborative interface for editing large knowledge bases. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 491–496.

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